IRIX Programmer's Reference Manual

Volume I

Version 5.0

Document Number 007-0602-050

IRIX Programmer's Reference Manual
Version 5.0
Document Number 007-0602-050

Silicon Graphics, Inc.
Mountain View, California

TABLE OF CONTENTS

2. System Calls

Table of Contents

NAME

intro – introduction to system calls and error numbers

SYNOPSIS

#include <errno.h>
#include <limits.h>

DESCRIPTION

This section describes all of the system calls. Most of these calls have one or more error returns. An error condition is indicated by an otherwise impossible returned value. This is almost always –1 or the NULL pointer; the individual descriptions specify the details. An error number is also made available in the external variable *errno*. *Errno* is not cleared on successful calls, so it should be tested only after an error has been indicated.

Many of these errors are caused by certain system or user limits being exceeded. In the individual manual pages, these limits are enclosed in braces (i.e. *{OPEN_MAX}*). The section *LIMITS* defines these limits, where and how they are configured, and whether they are alterable.

Each system call description attempts to list all possible error numbers. The following is a complete list of the error numbers and their names as defined in **<errno.h>**.

1 EPERM No permission match

Typically this error indicates an attempt to modify a file in some way forbidden except to its owner or super-user. It is also returned for attempts by ordinary users to do things allowed only to the super-user. A special case of this involves setuid/setgid shell scripts and a kernel that is configured with a non-zero value for *nosuidshells* (the shipped default). The kernel returns EPERM when a non-superuser attempts to execute such a shell script with a uid or gid which is different than the user's effective uid/gid.

2 ENOENT No such file or directory

This error occurs when a file name is specified and the file should exist but doesn't, or when one of the directories in a path name does not exist.

3 ESRCH No such process

No process can be found corresponding to that specified by *pid* in *kill*(2), *blockproc*(2), or *ptrace*(2).

4 EINTR Interrupted system call

An asynchronous signal (such as interrupt or quit), which the user has elected to catch, occurred during a system call. If execution is resumed after processing the signal, it will appear as if the interrupted system call returned this error condition [see section on

interruptibility].

5 EIO I/O error
Some physical I/O error has occurred. This error may in some cases occur on a call following the one to which it actually applies.

6 ENXIO No such device or address
I/O on a special file refers to a subdevice which does not exist, or beyond the limits of the device. It may also occur when, for example, a tape drive is not on-line or no disk pack is loaded on a drive.

7 E2BIG Arg list too long
An argument list longer than *{ARG_MAX}* bytes is presented to a member of the *exec*(2) family.

8 ENOEXEC Exec format error
A request is made to execute a file which, although it has the appropriate permissions, does not start with a valid magic number [see *a.out*(4)].

9 EBADF Bad file number
Either a file descriptor refers to no open file, or a *read*(2) [respectively, *write*(2)] request is made to a file which is open only for writing (respectively, reading).

10 ECHILD No child processes
A *wait* was executed by a process that had no existing or unwaited-for child processes.

11 EAGAIN Resource temporarily unavailable
1) A *fork* or *sproc* failed because the maximum number of processes system wide *{NPROC}* was exceeded or the user exceeded their limit on the number of child processes *{CHILD_MAX}*. 2) A system call failed because of insufficient memory or swap space. Later calls to the same routine may complete normally. 3) Some read system calls--involving empty streams or locked files/records--with the O_NDELAY flag set may return this error. See *read(2)*.

12 ENOMEM Not enough space
During an *exec*(2), *brk*(2), or *sbrk*(2), a process exceeds its maximum allowable size *{PROCSIZE_MAX}*. This may also be returned by device drivers if they cannot dynamically allocate enough space.

13 EACCES Permission denied
An attempt was made to access a file in a way forbidden by the protection system.

14 EFAULT Bad address
The system encountered a hardware fault in attempting to use an argument of a system call.

15 ENOTBLK Block device required
A non-block file was mentioned where a block device was required, e.g., in *mount*(2).

16 EBUSY Device or resource busy
An attempt was made to mount a device that was already mounted or an attempt was made to dismount a device on which there is an active file (open file, current directory, mounted-on file, active text segment). It will also occur if an attempt is made to enable accounting when it is already enabled. The device or resource is currently unavailable.

17 EEXIST File exists
An existing file was mentioned in an inappropriate context, e.g., *link*(2).

18 EXDEV Cross-device link
A link to a file on another device was attempted.

19 ENODEV No such device
An attempt was made to apply an inappropriate system call to a device; e.g., read a write-only device.

20 ENOTDIR Not a directory
A non-directory was specified where a directory is required, for example in a path prefix or as an argument to *chdir*(2).

21 EISDIR Is a directory
An attempt was made to write on a directory.

22 EINVAL Invalid argument
Some invalid argument (e.g., dismounting a non-mounted device; mentioning an undefined signal in *signal*(2) or *kill*(2); reading or writing a file for which *lseek*(2) has generated a negative pointer). Also set by the math functions described in the (3M) entries of this manual.

23 ENFILE Too many open files in system
The system file table is full (has exceeded *{NFILE_MAX}* entries), and temporarily no more *opens* can be accepted.

24 EMFILE Too many open files in a process
No process may have more than *{OPEN_MAX}* descriptors open at a time. Or the maximum number of shared memory segments *{SHMAT_MAX}* was exceeded.

25 ENOTTY Inappropriate IOCTL operation
An attempt was made to *ioctl*(2) a file that is not a special character device.

26 ETXTBSY Text file busy
An attempt was made to execute a pure-procedure program that is currently open for writing. Also an attempt to open for writing or to remove a pure-procedure program that is being executed.

27 EFBIG File too large
The size of a file exceeded the per process maximum file size *{FILESIZE_MAX}*.

28 ENOSPC No space left on device
During a *write*(2) to an ordinary file, there is no free space left on the device.

29 ESPIPE Illegal seek
An *lseek*(2) was issued to a pipe or FIFO.

30 EROFS Read-only file system
An attempt to modify a file or directory was made on a device mounted read-only.

31 EMLINK Too many links
An attempt to make more than the maximum number of links *{LINK_MAX}* to a file.

32 EPIPE Broken pipe
A write on a pipe or FIFO for which there is no process to read the data.

33 EDOM Argument out of range
The argument of a function in the math package (3M) or other library functions is out of the domain of the function.

34 ERANGE Result too large
The value of a function in the math package (3M) is not representable within machine precision, or a size specified is not large enough.

35 ENOMSG No message of desired type
An attempt was made to receive a message of a type that does not exist on the specified message queue [see *msgop*(2)].

36 EIDRM Identifier removed
This error is returned to processes that resume execution due to the removal of an identifier from the file system's name space [see *msgctl*(2), *semctl*(2), and *shmctl*(2)].

37-44 Reserved numbers

45 EDEADLK Deadlock situation detected/avoided
A deadlock situation was detected and avoided. This error pertains to file and record locking.

46 ENOLCK No record locks available
In *fcntl*(2) the setting or removing of record locks on a file cannot be accomplished because the system wide maximum number of record entries *{FLOCK_MAX}* has been exceeded.

50-57 Reserved numbers

60 ENOSTR Not a stream device
A *putmsg*(2) or *getmsg*(2) system call was attempted on a file descriptor that is not a STREAMS device.

61 ENODATA No data available

62 ETIME Timer expired
The timer set for a STREAMS *ioctl*(2) call has expired. The cause of this error is device specific and could indicate either a hardware or software failure, or perhaps a timeout value that is too short for the specific operation. The status of the *ioctl*(2) operation is indeterminate.

63 ENOSR Out of stream resources
During a STREAMS *open*(2), either no STREAMS queues or no STREAMS head data structures were available.

64 Reserved

65 ENOPKG Package not installed
This error occurs when users attempt to use a system call from a package which has not been installed.

66-70 Reserved numbers

71 EPROTO Protocol error
Some protocol error occurred. This error is device specific, but is generally not related to a hardware failure.

74-76 Reserved numbers

77 EBADMSG Not a data message
During a *read*(2), *getmsg*(2), or *ioctl*(2) I_RECVFD system call to a STREAMS device, something has come to the head of the queue that can't be processed. That something depends on the system call:
read(2) - control information or a passed file descriptor.
getmsg(2) - passed file descriptor.

ioctl(2) - control or data information.

80-82 Reserved numbers

83 ELIBACC Can not access a needed shared library
Trying to *exec*(2) an *a.out* that requires a shared library (to be linked in) and the shared library doesn't exist or the user doesn't have permission to use it.

84 ELIBBAD Accessing a corrupted shared library
Trying to *exec*(2) an *a.out* that requires a shared library (to be linked in) and *exec*(2) could not load the shared library. The shared library is probably corrupted.

85 ELIBSCN .lib section in *a.out* corrupted
Trying to *exec*(2) an *a.out* that requires a shared library (to be linked in) and there was erroneous data in the .lib section of the *a.out*. The .lib section tells *exec*(2) what shared libraries are needed. The *a.out* is probably corrupted.

86 ELIBMAX Attempting to link in more shared libraries than system limit
Trying to *exec*(2) an *a.out* that requires more shared libraries (to be linked in) than the system imposed maximum *{SHLIB_MAX}*.

87 ELIBEXEC Cannot exec a shared library directly
Trying to *exec*(2) a shared library directly. This is not allowed.

101 EWOULDBLOCK Operation would block
An operation which would cause a process to block was attempted on an object in non-blocking mode [see *ioctl* (2)].

102 EINPROGRESS Operation now in progress
An operation which takes a long time to complete (such as a *connect* (2)) was attempted on a non-blocking object [see *ioctl* (2)].

103 EALREADY Operation already in progress
An operation was attempted on a non-blocking object which already had an operation in progress.

104 ENOTSOCK Socket operation on non-socket

105 EDESTADDRREQ Destination address required
A required address was omitted from an operation on a socket.

106 EMSGSIZE Message too long
A message sent on a socket was larger than the internal message buffer.

107 EPROTOTYPE Protocol wrong type for socket
A protocol was specified which does not support the semantics of the socket type requested. For example, you cannot use the ARPA Internet UDP protocol with type SOCK_STREAM.

108 ENOPROTOOPT Option not supported by protocol
A bad option was specified in a *getsockopt*(2) or *setsockopt*(2) call.

109 EPROTONOSUPPORT Protocol not supported
The protocol has not been configured into the system or no implementation for it exists.

110 ESOCKTNOSUPPORT Socket type not supported
The support for the socket type has not been configured into the system or no implementation exists.

111 EOPNOTSUPP Operation not supported on socket
For example, trying to *accept* a connection on a datagram socket.

112 EPFNOSUPPORT Protocol family not supported
The protocol family has not been configured into the system or no implementation exists.

113 EAFNOSUPPORT Address family not supported by protocol family
An address incompatible with the requested protocol was used. For example, you shouldn't necessarily expect to be able to use PUP Internet addresses with ARPA Internet protocols.

114 EADDRINUSE Address already in use
Only one usage of each address is normally permitted.

115 EADDRNOTAVAIL Can't assign requested address
Normally results from an attempt to create a socket with an address not on this machine.

116 ENETDOWN Network is down
A socket operation encountered a dead network.

117 ENETUNREACH Network is unreachable
A socket operation was attempted to an unreachable network.

118 ENETRESET Network dropped connection on reset
The host you were connected to crashed and rebooted.

119 ECONNABORTED Software caused connection abort
A connection abort was caused internal to your host machine.

120 ECONNRESET Connection reset by peer
A connection was forcibly closed by a peer. This normally results from a peer executing a *shutdown* (2) call.

121 ENOBUFS No buffer space available
An operation on a socket or pipe performed because the system lacked sufficient buffer space.

122 EISCONN Socket is already connected
A *connect* request was made on an already connected socket; or a *sendto* or *sendmsg* request on a connected socket specified a destination other than the connected party.

123 ENOTCONN Socket is not connected
A request to send or receive data was disallowed because the socket was not connected.

124 ESHUTDOWN Can't send after socket shutdown
A request to send data was disallowed because the socket had already been shut down with a previous *shutdown* (2) call.

125 ETOOMANYREFS Too many references: can't splice

126 ETIMEDOUT Connection timed out
A *connect* request failed because the connected party did not properly respond after a period of time. (The timeout period is dependent on the communication protocol.) [see section on interruptibility.]

127 ECONNREFUSED Connection refused
No connection could be made because the target machine actively refused it.

128 EHOSTDOWN Host is down

129 EHOSTUNREACH No route to host

130 ELOOP Too many levels of symbolic links
A path name lookup involved more than 8 symbolic links.

131 ENAMETOOLONG File name too long
A component of a path name exceeded *{NAME_MAX}* characters, or an entire path name exceeded *{PATH_MAX}* characters.

132 ENOTEMPTY Directory not empty

133 EDQUOT Disc quota exceeded

134 ESTALE Stale NFS file handle

135 ENFSREMOTE Too many levels of remote in path

DEFINITIONS

Process ID Each active process in the system is uniquely identified by a positive integer called a process ID. The range of this ID is from 1 to *{PID_MAX}* (30,000).

Parent Process ID A new process is created by a currently active process [see *fork*(2) and *sproc*(2)]. The parent process ID of a process is the process ID of its creator.

Process Group ID Each active process is a member of a process group that is identified by a positive integer called the process group ID. This ID is the process ID of the group leader. This grouping permits the signaling of related processes [see *kill*(2)].

Tty Group ID Each active process can be a member of a terminal group that is identified by a positive integer called the tty group ID. This grouping is used to terminate a group of related processes upon termination of one of the processes in the group [see *exit*(2) and *signal*(2)].

Real User ID and Real Group ID Each user allowed on the system is identified by a positive integer 0 to *{UID_MAX}* (60,000) called a real user ID.

Each user is also a member of a group. The group is identified by a positive integer called the real group ID.

An active process has a real user ID and real group ID that are set to the real user ID and real group ID, respectively, of the user responsible for the creation of the process.

Effective User ID and Effective Group ID An active process has an effective user ID and an effective group ID that are used to determine file access permissions (see below). The effective user ID and effective group ID are equal to the process's real user ID and real group ID respectively, unless the process or one of its ancestors evolved from a file that had the set-user-ID bit or set-group ID bit set [see *exec*(2)].

Supplementary Group ID A process has up to *{NGROUPS_MAX}* supplementary group IDs used in determining file access permissions, in addition to the effective group ID. The supplementary group IDs of a process are set

to the supplementary group IDs of the parent process when the process is created.

Super-user A process is recognized as a *super-user* process and is granted special privileges, such as immunity from file permissions, if its effective user ID is 0.

Special Processes The processes with a process ID of 0 and a process ID of 1 are special processes and are referred to as *proc0* and *proc1*.

Proc0 is the scheduler. *Proc1* is the initialization process (*init*). Proc1 is the ancestor of every other process in the system and is used to control the process structure.

File Descriptor A file descriptor is a small integer used to do I/O on a file. The value of a file descriptor is from 0 to (NOFILES - 1). A process may have no more than NOFILES file descriptors open simultaneously. A file descriptor is returned by system calls such as *open*(2), or *pipe*(2). The file descriptor is used as an argument by calls such as *read*(2), *write*(2), *ioctl*(2), *mmap*(2), *munmap*(2), and *close*(2). NOFILES is a synonym for *{OPEN_MAX}*.

File Name Names consisting of 1 to *{NAME_MAX}* characters may be used to name an ordinary file, special file or directory.

These characters may be selected from the set of all character values excluding \0 (null) and the ASCII code for / (slash).

Note that it is generally unwise to use *, ?, [, or] as part of file names because of the special meaning attached to these characters by the shell [see *sh*(1)]. Although permitted, the use of unprintable characters in file names should be avoided.

Path Name and Path Prefix A path name is a null-terminated character string starting with an optional slash (/), followed by zero or more directory names separated by slashes, optionally followed by a file name. The entire length of a path name is limited to *{PATH_MAX}* characters.

If a path name begins with a slash, the path search begins at the *root* directory. Otherwise, the search begins from the current working directory.

A slash by itself names the root directory.

Unless specifically stated otherwise, the null path name is treated as if it named a non-existent file.

Directory Directory entries are called links. By convention, a directory contains at least two links, . and .., referred to as *dot* and *dot-dot* respectively. Dot refers to the directory itself and dot-dot refers to its parent directory.

Root Directory and Current Working Directory Each process has associated with it a concept of a root directory and a current working directory for the purpose of resolving path name searches. The root directory of a process need not be the root directory of the root file system.

File Access Permissions Read, write, and execute/search permissions on a file are granted to a process if one or more of the following are true:

The effective user ID of the process is super-user.

The effective user ID of the process matches the user ID of the owner of the file and the appropriate access bit of the "owner" portion (0700) of the file mode is set.

The effective user ID of the process does not match the user ID of the owner of the file, and the effective group ID of the process matches the group of the file and the appropriate access bit of the "group" portion (0070) of the file mode is set.

The effective user ID of the process does not match the user ID of the owner of the file, and the effective group ID of the process does not match the group ID of the file, and the appropriate access bit of the "other" portion (0007) of the file mode is set.

Otherwise, the corresponding permissions are denied.

Message Queue Identifier A message queue identifier (msqid) is a unique positive integer created by a *msgget*(2) system call. Each msqid has a message queue and a data structure associated with it. The data structure is referred to as *msqid_ds* and contains the following members:

```
struct  ipc_perm msg_perm;
struct  msg *msg_first;
struct  msg *msg_last;
ushort  msg_cbytes;
ushort  msg_qnum;
ushort  msg_qbytes;
ushort  msg_lspid;
```

```
ushort    msg_lrpid;
time_t    msg_stime;
time_t    msg_rtime;
time_t    msg_ctime;
```

msg_perm is an ipc_perm structure that specifies the message operation permission (see below). This structure includes the following members:

```
ushort    cuid;       /* creator user id */
ushort    cgid;       /* creator group id */
ushort    uid;        /* user id */
ushort    gid;        /* group id */
ushort    mode;       /* r/w permission */
ushort    seq;        /* slot usage sequence # */
key_t     key;        /* key */
```

msg *msg_first
is a pointer to the first message on the queue.

msg *msg_last
is a pointer to the last message on the queue.

msg_cbytes
is the current number of bytes on the queue.

msg_qnum
is the number of messages currently on the queue.

msg_qbytes
is the maximum number of bytes allowed on the queue.

msg_lspid
is the process id of the last process that performed a *msgsnd* operation.

msg_lrpid
is the process id of the last process that performed a *msgrcv* operation.

msg_stime
is the time of the last *msgsnd* operation.

msg_rtime
is the time of the last *msgrcv* operation

msg_ctime
is the time of the last *msgctl*(2) operation that changed a member of the above structure.

Message Operation Permissions In the *msgop*(2) and *msgctl*(2) system call descriptions, the permission required for an operation is given as "{token}", where "token" is the type of permission needed, interpreted as follows:

00400	Read by user
00200	Write by user
00040	Read by group
00020	Write by group
00004	Read by others
00002	Write by others

Read and write permissions on a msqid are granted to a process if one or more of the following are true:

The effective user ID of the process is super-user.

The effective user ID of the process matches **msg_perm.cuid** or **msg_perm.uid** in the data structure associated with *msqid* and the appropriate bit of the ''user'' portion (0600) of **msg_perm.mode** is set.

The effective group ID of the process matches **msg_perm.cgid** or **msg_perm.gid** and the appropriate bit of the ''group'' portion (060) of **msg_perm.mode** is set.

The appropriate bit of the ''other'' portion (006) of **msg_perm.mode** is set.

Otherwise, the corresponding permissions are denied.

Semaphore Identifier A semaphore identifier (semid) is a unique positive integer created by a *semget*(2) system call. Each semid has a set of semaphores and a data structure associated with it. The data structure is referred to as *semid_ds* and contains the following members:

```
struct  ipc_perm sem_perm;  /* operation permission struct */
struct  sem *sem_base;      /* ptr to first semaphore in set */
ushort  sem_nsems;          /* number of sems in set */
time_t  sem_otime;          /* last operation time */
time_t  sem_ctime;          /* last change time */
                            /* Times measured in secs since */
                            /* 00:00:00 GMT, Jan. 1, 1970 */
```

sem_perm is an ipc_perm structure that specifies the semaphore operation permission (see below). This structure includes the following members:

```
ushort   uid;    /* user id */
ushort   gid;    /* group id */
ushort   cuid;   /* creator user id */
ushort   cgid;   /* creator group id */
ushort   mode;   /* r/a permission */
ushort   seq;    /* slot usage sequence number */
key_t    key;    /* key */
```

sem_nsems
is equal to the number of semaphores in the set. Each semaphore in the set is referenced by a positive integer referred to as a *sem_num*. Sem_num values run sequentially from 0 to the value of sem_nsems minus 1.

sem_otime
is the time of the last *semop*(2) operation.

sem_ctime
is the time of the last *semctl*(2) operation that changed a member of the above structure.

A semaphore is a data structure called *sem* that contains the following members:

```
ushort   semval;   /* semaphore value */
short    sempid;   /* pid of last operation */
ushort   semncnt;  /* # awaiting semval > cval */
ushort   semzcnt;  /* # awaiting semval = 0 */
```

semval is a non-negative integer which is the actual value of the semaphore.

sempid is equal to the process ID of the last process that performed a semaphore operation on this semaphore.

semncnt
is a count of the number of processes that are currently suspended awaiting this semaphore's semval to become greater than its current value.

semzcnt
is a count of the number of processes that are currently suspended awaiting this semaphore's semval to become zero.

Semaphore Operation Permissions In the *semop*(2) and *semctl*(2) system call descriptions, the permission required for an operation is given as "{token}", where "token" is the type of permission needed interpreted as follows:

00400	Read by user
00200	Alter by user
00040	Read by group
00020	Alter by group
00004	Read by others
00002	Alter by others

Read and alter permissions on a semid are granted to a process if one or more of the following are true:

The effective user ID of the process is super-user.

The effective user ID of the process matches **sem_perm.cuid** or **sem_perm.uid** in the data structure associated with *semid* and the appropriate bit of the ''user'' portion (0600) of **sem_perm.mode** is set.

The effective group ID of the process matches **sem_perm.cgid** or **sem_perm.gid** and the appropriate bit of the ''group'' portion (060) of **sem_perm.mode** is set.

The appropriate bit of the ''other'' portion (006) of **sem_perm.mode** is set.

Otherwise, the corresponding permissions are denied.

Shared Memory Identifier A shared memory identifier (shmid) is a unique positive integer created by a *shmget*(2) system call. Each shmid has a segment of memory (referred to as a shared memory segment) and a data structure associated with it. (Note that these shared memory segments must be explicitly removed by the user after the last reference to them is removed.) The data structure is referred to as *shmid_ds* and contains the following members:

```
struct  ipc_perm shm_perm;  /* operation permission struct */
int     shm_segsz;          /* size of segment */
struct  region *shm_reg;    /*ptr to region structure */
char    pad[4];             /* for swap compatibility */
ushort  shm_lpid;           /* pid of last operation */
ushort  shm_cpid;           /* creator pid */
ushort  shm_nattch;         /* number of current attaches */
ushort  shm_cnattch;        /* used only for shminfo */
```

```
time_t   shm_atime;    /* last attach time */
time_t   shm_dtime;    /* last detach time */
time_t   shm_ctime;    /* last change time */
                       /* Times measured in secs since */
                       /* 00:00:00 GMT, Jan. 1, 1970 */
```

shm_perm is an ipc_perm structure that specifies the shared memory operation permission (see below). This structure includes the following members:

```
ushort   cuid;    /* creator user id */
ushort   cgid;    /* creator group id */
ushort   uid;     /* user id */
ushort   gid;     /* group id */
ushort   mode;    /* r/w permission */
ushort   seq;     /* slot usage sequence # */
key_t    key;     /* key */
```

shm_segsz
specifies the size of the shared memory segment in bytes.

shm_cpid
is the process id of the process that created the shared memory identifier.

shm_lpid
is the process id of the last process that performed a *shmop*(2) operation.

shm_nattch
is the number of processes that currently have this segment attached.

shm_atime
is the time of the last *shmat*(2) operation,

shm_dtime
is the time of the last *shmdt*(2) operation.

shm_ctime
is the time of the last *shmctl*(2) operation that changed one of the members of the above structure.

Shared Memory Operation Permissions In the *shmop*(2) and *shmctl*(2) system call descriptions, the permission required for an operation is given as "{token}", where "token" is the type of permission needed interpreted as follows:

00400	Read by user
00200	Write by user
00040	Read by group
00020	Write by group
00004	Read by others
00002	Write by others

Read and write permissions on a shmid are granted to a process if one or more of the following are true:

The effective user ID of the process is super-user.

The effective user ID of the process matches **shm_perm.cuid** or **shm_perm.uid** in the data structure associated with *shmid* and the appropriate bit of the "user" portion (0600) of **shm_perm.mode** is set.

The effective group ID of the process matches **shm_perm.cgid** or **shm_perm.gid** and the appropriate bit of the "group" portion (060) of **shm_perm.mode** is set.

The appropriate bit of the "other" portion (06) of **shm_perm.mode** is set.

Otherwise, the corresponding permissions are denied.

STREAMS A set of kernel mechanisms that support the development of network services and data communication *drivers*. It defines interface standards for character input/output within the kernel and between the kernel and user level processes. The STREAMS mechanism is composed of utility routines, kernel facilities and a set of data structures.

Stream A stream is a full-duplex data path within the kernel between a user process and driver routines. The primary components are a *stream head*, a *driver* and zero or more *modules* between the *stream head* and *driver*. A *stream* is analogous to a Shell pipeline except that data flow and processing are bidirectional.

Stream Head In a *stream*, the *stream head* is the end of the *stream* that provides the interface between the *stream* and a user process. The principle functions of the *stream head* are processing STREAMS-related system calls, and passing data and information between a user process and the *stream*.

Driver In a *stream*, the *driver* provides the interface between peripheral hardware and the *stream*. A *driver* can also be a pseudo-*driver*, such as a *multiplexor* or log *driver* [see *log*(7)], which is not associated with a

hardware device.

Module A module is an entity containing processing routines for input and output data. It always exists in the middle of a *stream*, between the stream's head and a *driver*. A *module* is the STREAMS counterpart to the commands in a Shell pipeline except that a module contains a pair of functions which allow independent bidirectional (*downstream* and *upstream*) data flow and processing.

Downstream In a *stream*, the direction from *stream head* to *driver*.

Upstream In a *stream*, the direction from *driver* to *stream head*.

Message In a *stream*, one or more blocks of data or information, with associated STREAMS control structures. *Messages* can be of several defined types, which identify the *message* contents. *Messages* are the only means of transferring data and communicating within a *stream*.

Message Queue In a *stream*, a linked list of *messages* awaiting processing by a *module* or *driver*.

Read Queue In a *stream*, the *message queue* in a *module* or *driver* containing *messages* moving *upstream*.

Write Queue In a *stream*, the *message queue* in a *module* or *driver* containing *messages* moving *downstream*.

Multiplexor A multiplexor is a driver that allows *streams* associated with several user processes to be connected to a single *driver*, or several *drivers* to be connected to a single user process. STREAMS does not provide a general multiplexing *driver*, but does provide the facilities for constructing them, and for connecting multiplexed configurations of *streams*.

LIMITS

The various limits can be categorized as follows: not modifiable, modifiable at sysgen time and modifiable at runtime. Limits may apply system wide or per process. Most per process limits are inherited on *fork*, *sproc*, and *exec*.

{NPROC} is the maximum number of processes allowed to run concurrently on the system. It is modifiable at sysgen time and is located in the file */usr/sysgen/master.d/kernel*. The current number in use and the current configured number may be retrieved via the *sar -v* command.

{NFILE_MAX} is the maximum number of open files that can be simultaneously active system wide. It is modifiable at sysgen time and is defined by the parameter **NFILE** in the file */usr/sysgen/master.d/kernel*. The current number in use and the current configured number may be retrieved via the *sar -v* command.

{FLOCK_MAX} is the maximum number of file locks system wide that may be active. It is modifiable at sysgen time and is defined by the parameter **FLCKREC** in the file */usr/sysgen/master.d/kernel*. The current number in use and the current configured number may be retrieved via the *sar -v* command.

{NAME_MAX} is the maximum length of a file name. It is defined in *limits.h* and is not modifiable.

{PATH_MAX} is the maximum length of a path name. It is defined in *limits.h* and is not modifiable.

{LINK_MAX} is the maximum number of hard links that may be made to a given file. It is defined in *limits.h* and is not modifiable.

{OPEN_MAX} is the maximum number of open files a given process may have. The minimum value this can have is defined in *limits.h*. It can be changed at sysgen time by changing the parameter **NOFILES** in the file */usr/sysgen/master.d/kernel*. Note that 100 is the maximum. The current number of open files configured may be retrieved by either *getdtablesize*(2), *sysconf*(2), or *ulimit*(2).

{CHILD_MAX} is the maximum number of processes a given user may have running simultaneously. The minimum value this can have is defined in *limits.h*. It can be changed at sysgen time by changing the parameter **MAXUP** in the file */usr/sysgen/master.d/kernel*. The current configured maximum may be obtained from *sysconf*(2).

{SHLIB_MAX} is the maximum number of shared libraries a program can link with. It can be changed at sysgen time by changing the parameter **SHLBMAX** in the file */usr/sysgen/master.d/kernel*.

{ARG_MAX} is the maximum number of bytes that may be passed via *exec*. The minimum value this can have is defined in *limits.h*. It can be changed at sysgen time by changing the parameter **SYSNCARGS** in the file */usr/sysgen/master.d/kernel*. The current configured maximum may be obtained from *sysconf*(2).

{FILESIZE_MAX} is the maximum size in bytes that a single file can grow to. The default maximum can be changed at sysgen time by changing the contents of the *defrlimit* structure in the file */usr/sysgen/master.d/kernel*. This structure defines both the current and maximum size. A process can change its current maximum file size up to the defined maximum via *setrlimit*(2). The current file size may be obtained by a program via *getrlimit*(2) or *ulimit*(2); and may be obtained from the shell by either the *limit* or *ulimit* built-in command.

{PROCSIZE_MAX} is the maximum virtual size a process can grow to. A process is made up an arbitrary number of virtual spaces. There are limits on the total size of process as well as certain limits on individual spaces. The overall limit is defined by the parameter **MAXUMEM** in the file */usr/sysgen/master.d/kernel*. It may be changed at sysgen time. The maximum stack size is defined by the **RLIM_STACK** parameter in the *defrlimit* structure in */usr/sysgen/master.d/kernel*. The maximum data size is defined by the **RLIM_DATA** parameter in the *defrlimit* structure in */usr/sysgen/master.d/kernel*. The maximum size of a shared memory segment is defined by the **SHMMAX** parameter in */usr/sysgen/master.d/shm*.

{SHMSEG_MAX} is the maximum number of shared memory segments system wide. It can be changed at sysgen time by changing the parameter **SHMMNI** in the file */usr/sysgen/master.d/shm*.

{SHMAT_MAX} is the maximum number of shared memory segments a given process may attach to. It can be changed at sysgen time by changing the parameter **SHMSEG** in the file */usr/sysgen/master.d/shm*.

{PLOCK_MAX} is the maximum number of pages a non-privileged process is allowed to lock at any one time. It can be changed at sysgen time by changing the parameter **MAXLKMEM** in the file */usr/sysgen/master.d/kernel*.

INTERRUPTIBILITY

Certain system calls can be interrupted by the process receiving a signal. These include but are not limited to *open*(2), *read*(2), *write*(2), and *ioctl*(2). The conditions under which a given system call can be interrupted are listed on each manual page. In addition, any file system oriented system call can either time out (ETIMEDOUT or be interrupted EINTR) if the object of the system call is located on a remote system accessed via NFS. Whether these system calls can time out or be interrupted is based on how the underlying file system was mounted on the local machine [see *fstab*(4) and *mount*(1M)].

SEE ALSO

sar(1), lboot(1M), mount(1M), getrlimit(2), pathconf(2), sysconf(2), **ulimit**(2), intro(3), perror(3), fstab(4).

NAME

accept – accept a connection on a socket

SYNOPSIS

```
#include <sys/types.h>
#include <sys/socket.h>

ns = accept(s, addr, addrlen)
int ns, s;
struct sockaddr *addr;
int *addrlen;
```

DESCRIPTION

The argument *s* is a socket that has been created with *socket*(2), bound to an address with *bind*(2), and is listening for connections after a *listen*(2). *Accept* extracts the first connection on the queue of pending connections, creates a new socket with the same properties of *s* and allocates a new file descriptor, *ns*, for the socket. If no pending connections are present on the queue, and the socket is not marked as non-blocking, *accept* blocks the caller until a connection is present. If the socket is marked non-blocking and no pending connections are present on the queue, *accept* returns an error as described below. The accepted socket, *ns*, may not be used to accept more connections. The original socket *s* remains open.

If *addr* is non-zero, it is a result parameter that is filled in with the address of the connecting entity, as known to the communications layer. The exact format of the *addr* parameter is determined by the domain in which the communication is occurring. The *addrlen* is a value-result parameter. It should initially contain the amount of space pointed to by *addr*; on return it will contain the actual length (in bytes) of the address returned. If *addr* is zero, *addrlen* is ignored.

This call is used with connection-based socket types, currently with SOCK_STREAM.

It is possible to *select*(2) a socket for the purposes of doing an *accept* by selecting it for read.

RETURN VALUE

The call returns –1 on error. If it succeeds, it returns a non-negative integer that is a descriptor for the accepted socket.

ERRORS

The *accept* will fail if:

[EBADF] The descriptor is invalid.

[ENOTSOCK]	The descriptor references a file, not a socket.
[EOPNOTSUPP]	The referenced socket is not of type SOCK_STREAM.
[EFAULT]	The *addr* or *addrlen* parameter is not in a writable part of the user address space.
[EWOULDBLOCK]	The socket is marked non-blocking and no connections are present to be accepted.
[EMFILE]	The per-process descriptor table is full.
[ENFILE]	The system file table is full.

SEE ALSO

bind(2), connect(2), listen(2), select(2), socket(2)

NAME

access – determine accessibility of a file

C SYNOPSIS

#include <unistd.h>

int access (const char *path, int amode);

DESCRIPTION

Path points to a path name naming a file. *access* checks the named file for accessibility according to the bit pattern contained in *amode* using the real user ID in place of the effective user ID and the group access list (including the real group ID) in place of the effective group ID. The bit pattern contained in *amode* is constructed as follows:

R_OK	04	read
W_OK	02	write
X_OK	01	execute (search)
F_OK	00	check existence of file

access will fail if one or more of the following are true:

[ENOTDIR]	A component of the path prefix is not a directory.
[ENOENT]	Read, write, or execute (search) permission is requested for a null path name.
[ENOENT]	The named file does not exist.
[EACCES]	Search permission is denied on a component of the path prefix.
[ENAMETOOLONG]	The length of *path* exceeds *{PATH_MAX}*, or a pathname component is longer than *{NAME_MAX}*.
[ELOOP]	Too many symbolic links were encountered in translating the pathname.
[EROFS]	Write access is requested for a file on a read-only file system.
[ETXTBSY]	Write access is requested for a pure procedure (shared text) file that is being executed.
[EACCES]	Permission bits of the file mode do not permit the requested access.
[EFAULT]	*Path* points outside the allocated address space for the process.

The owner of a file has permission checked with respect to the "owner" read, write, and execute mode bits. Members of the file's group other than the owner have permissions checked with respect to the "group" mode bits, and all others have permissions checked with respect to the "other" mode bits.

SEE ALSO

chmod(2), stat(2).

DIAGNOSTICS

If the requested access is permitted, a value of 0 is returned. Otherwise, a value of −1 is returned and *errno* is set to indicate the error.

NAME

acct – enable or disable process accounting

C SYNOPSIS

int acct (path)
char *path;

DESCRIPTION

acct is used to enable or disable the system process accounting routine. If the routine is enabled, an accounting record will be written on an accounting file for each process that terminates. Termination can be caused by one of two things: an *exit* call or a signal [see *exit*(2) and *signal*(2)]. The effective user ID of the calling process must be superuser to use this call.

path points to a pathname naming the accounting file. The accounting file format is given in *acct*(4).

The accounting routine is enabled if *path* is non-zero and no errors occur during the system call. It is disabled if *path* is zero and no errors occur during the system call.

acct will fail if one or more of the following are true:

[EPERM]	The effective user of the calling process is not superuser.
[EBUSY]	An attempt is being made to enable accounting when it is already enabled.
[ENOTDIR]	A component of the path prefix is not a directory.
[ENOENT]	One or more components of the accounting file pathname do not exist.
[EACCES]	The file named by *path* is not an ordinary file.
[EROFS]	The named file resides on a read-only file system.
[EFAULT]	*path* points to an illegal address.

SEE ALSO

exit(2), signal(2), acct(4).

DIAGNOSTICS

Upon successful completion, a value of 0 is returned. Otherwise, a value of −1 is returned and *errno* is set to indicate the error.

NAME

adjtime – correct the time to allow synchronization of the system clock

SYNOPSIS

#include <sys/time.h>

adjtime(delta, olddelta)
struct timeval *delta;
struct timeval *olddelta;

DESCRIPTION

Adjtime makes small adjustments to the system time, as returned by *gettimeofday*(3B), advancing or retarding it by the time specified by the timeval *delta*. If *delta* is negative, the clock is slowed down by incrementing it more slowly than normal until the correction is complete. If *delta* is positive, a larger increment than normal is used. The skew used to perform the correction is generally a fraction of one percent. Thus, the time is always a monotonically increasing function. A time correction from an earlier call to *adjtime* may not be finished when *adjtime* is called again. If *olddelta* is non-zero, then the structure pointed to will contain, upon return, the number of microseconds still to be corrected from the earlier call.

All of the significant digits in *delta* are used by *adjtime*, despite the fact that the system clock is incremented much less frequently than once each microsecond.

This call may be used by time servers that synchronize the clocks of computers in a local area network. Such time servers would slow down the clocks of some machines and speed up the clocks of others to bring them to the average network time.

Permanent adjustment of the rate of the system clock can be accomplished by modifying the *timetrim* parameter in the *master.d/kernel* file. This parameter is used by the operating system to compensate for variations among machines. It can be used to substantially improve the accuracy of the system clock. The *timetrim* parameter can also be changed with *syssgi*(2).

The use of *adjtime*(2) is restricted to the super-user.

RETURN VALUE

A return value of 0 indicates that the call succeeded. A return value of –1 indicates that an error occurred, and in this case an error code is stored in the global variable *errno*.

ERRORS

The following error codes may be set in *errno*:

[EFAULT] An argument points outside the process's allocated address space.

[EPERM] The process's effective user ID is not that of the super-user.

SEE ALSO

date(1), gettimeofday(3B), syssgi(2), timed(1M), timedc(1M), timeslave(1M),

NAME

alarm – set a process alarm clock

SYNOPSIS

#include <unistd.h>

unsigned alarm (unsigned sec);

DESCRIPTION

alarm instructs the alarm clock of the calling process to send the signal **SIGALRM** to the calling process after the number of real time seconds specified by *sec* have elapsed [see *signal*(2)].

Alarm requests are not stacked; successive calls reset the alarm clock of the calling process.

If *sec* is 0, any previously made alarm request is canceled.

SEE ALSO

getitimer(2), pause(2), signal(2), sigpause(2), sigset(2), sigaction(2), sigsuspend(2), sigprocmask(2), sigvec(3B), signal(3B), sigpause(3B).

DIAGNOSTICS

alarm returns the amount of time previously remaining in the alarm clock of the calling process.

BUGS

alarm calls reset the itimer clock of the calling process.

NAME

bind – bind a name to a socket

SYNOPSIS

#include <sys/types.h>
#include <sys/socket.h>

bind(s, name, namelen)
int s;
struct sockaddr *name;
int namelen;

DESCRIPTION

Bind assigns a name to an unnamed socket. When a socket is created with *socket*(2) it exists in a name space (address family) but has no name assigned. *Bind* requests that *name* be assigned to the socket.

The rules used in name binding vary between communication domains. Consult the protocol manual entries in section 7 for detailed information.

RETURN VALUE

If the bind is successful, a 0 value is returned. A return value of –1 indicates an error, which is further specified in the global *errno*.

ERRORS

The *bind* call will fail if:

[EBADF]	*S* is not a valid descriptor.
[ENOTSOCK]	*S* is not a socket.
[EADDRNOTAVAIL]	The specified address is not available from the local machine.
[EADDRINUSE]	The specified address is already in use.
[EINVAL]	The socket is already bound to an address.
[EACCES]	The requested address is protected, and the current user has inadequate permission to access it.
[EFAULT]	The *name* parameter is not in a valid part of the user address space.

SEE ALSO

connect(2), listen(2), socket(2)

NAME

blockproc, unblockproc, setblockproccnt, blockprocall, unblockprocall, setblockproccntall – routines to block/unblock processes

C SYNOPSIS

#include <sys/types.h>
#include <sys/prctl.h>

int blockproc (pid_t pid)

int unblockproc (pid_t pid)

int setblockproccnt (pid_t pid, int count)

int blockprocall (pid_t pid)

int unblockprocall (pid_t pid)

int setblockproccntall (pid_t pid, int count)

DESCRIPTION

These routines provide a complete set of blocking/unblocking capabilities for processes. Blocking is implemented with a counting semaphore in the kernel. Each call to *blockproc* decrements the count. When the count becomes negative, the process is suspended. When *unblockproc* is called, the count is incremented. If the count becomes non-negative (>= 0), the process is restarted. This provides both a simple, race free synchronization ability between two processes and a much more powerful capability to synchronize multiple processes.

In order to guarantee a known starting place, the *setblockproccnt* function may be called, which will force the semaphore count to the value given by *count*. New processes have their semaphore zeroed. Normally, *count* should be set to 0. If the resulting block count is greater than or equal to zero and the process is currently blocked, it will be unblocked. If the resulting block count is less than zero, the process will be blocked. Using this, a simple rendezvous mechanism can be set up. If one process wants to wait for *n* other processes to complete, it could set its block count to *-n*. This would immediately force the process to block. Then as each process finishes, it unblocks the waiting process. When the *n'th* process finishes the waiting process will be awakened.

The *blockprocall, unblockprocall,* and *setblockproccntall* system calls perform the same actions as *blockproc, unblockproc,* and *setblockproccnt,* respectively, but act on all processes in the given process' share group. A share group is a group of processes created with the *sproc(2)* system call. If a process does not belong to a share group, the effect of the plural form of a call will be the same as that of the singular form.

A process may block another provided that standard UNIX permissions are satisfied.

A process may determine whether another is blocked by using the *prctl(2)* system call. It should be noted that since other processes may unblock the subject process at any time, the answer should be interpreted as a snapshot only.

These routines will fail and no operation will be performed if one or more of the following are true:

[ESRCH] The *pid* specified does not exist.

[EPERM] The caller is not operating on itself, its effective user ID is not super-user, and its real or effective user ID does not match the real or effective user ID of the target process.

[EINVAL] The count value that would result from the requested *blockproc*, *unblockproc* or *setblockproccnt* is less than **PR_MINBLOCKCNT** or greater than **PR_MAXBLOCKCNT** as defined in *sys/prctl.h*.

SEE ALSO

sproc(2), prctl(2).

DIAGNOSTICS

Upon successful completion, 0 is returned. Otherwise, a value of −1 is returned to the calling process, and *errno* is set to indicate the error. When using the *blockprocall, unblockprocall,* and *setblockproccntall* calls, an error may occur on any of the processes in the share group. These calls will attempt to perform the given action on each process in the share group despite earlier errors, and set *errno* to indicate the error of the last failure to occur.

NAME

brk, sbrk – change data segment space allocation

C SYNOPSIS

int brk (endds)
void *endds;

void *sbrk (incr)
int incr;

DESCRIPTION

brk and *sbrk* are used to change dynamically the amount of space allocated for the calling process's data segment [see *exec*(2)]. The change is made by resetting the process's break value and allocating the appropriate amount of space. The break value is the address of the first location beyond the end of the data segment. The amount of allocated space increases as the break value increases. Newly allocated space is set to zero. If, however, the same memory space is reallocated to the same process its contents are undefined.

brk sets the break value to *endds* and changes the allocated space accordingly.

Sbrk adds *incr* bytes to the break value and changes the allocated space accordingly. *Incr* can be negative, in which case the amount of allocated space is decreased.

brk and *sbrk* will fail without making any change in the allocated space if one or more of the following are true:

[ENOMEM] Such a change would result in more space being allocated than is allowed by the system-imposed maximum process size *{PROCSIZE_MAX}*. [see *intro*(2)].

[EAGAIN] There is insufficient amount of operating system memory to hold the data structures needed to describe the requested space. This is likely a temporary failure.

SEE ALSO

exec(2), intro(2), shmop(2), getrlimit(2), ulimit(2), end(3C).

DIAGNOSTICS

Upon successful completion, *brk* returns a value of 0 and *sbrk* returns the old break value. Otherwise, a value of −1 is returned and *errno* is set to indicate the error.

NAME

cacheflush – flush contents of instruction and/or data cache

SYNOPSIS

#include <sys/cachectl.h>

cacheflush (void *addr, int nbytes, int cache)

DESCRIPTION

Flushes contents of indicated cache(s) for user addresses in the range *addr* to *addr+nbytes*-1). The *cache* parameter may be one of:

ICACHE	Flush only the instruction cache
DCACHE	Flush only the data cache
BCACHE	Flush both the instruction and the data cache

RETURN VALUE

cacheflush returns 0 when no errors are detected. If errors are detected, *cachectl* returns –1 with the error cause indicated in *errno*.

ERRORS

[EINVAL]	The *cache* parameter is not one of ICACHE, DCACHE, or BCACHE.
[EFAULT]	Some or all of the address range *addr* to (*addr+nbytes*-1) is not accessible.

NAME

cachectl – mark pages cacheable or uncacheable

SYNOPSIS

#include <sys/cachectl.h>

cachectl (void *addr, int nbytes, int op)

DESCRIPTION

The *cachectl* system call allows a process to make ranges of its address space cacheable or uncacheable. Initially, a process's entire address space is cacheable.

The *op* parameter may be one of:

CACHEABLE Make the indicated pages cacheable

UNCACHEABLE Make the indicated pages uncacheable

The CACHEABLE and UNCACHEABLE *op*'s affect the address range indicated by *addr* and *nbytes*. *addr* must be page aligned and nbytes must be a multiple of the page size.

Changing a page from UNCACHEABLE state to CACHEABLE state will cause both the instruction and data caches to be flushed if necessary to avoid stale cache information.

RETURN VALUE

cachectl returns 0 when no errors are detected. If errors are detected, *cachectl* returns –1 with the error cause indicated in *errno*.

ERRORS

[EINVAL]	The *op* parameter is not one of CACHEABLE or UNCACHEABLE.
[EINVAL]	The *addr* parameter is not page aligned, or *nbytes* is not a multiple of the page size.
[EFAULT]	Some or all of the address range *addr* to (*addr*+*nbytes*-1) is not accessible.

NAME

chdir – change working directory

C SYNOPSIS

#include <unistd.h>

int chdir (const char *path);

DESCRIPTION

Path points to the path name of a directory. *chdir* causes the named directory to become the current working directory, the starting point for path searches for path names not beginning with /.

chdir will fail and the current working directory will be unchanged if one or more of the following are true:

[ENOTDIR]	A component of the path name is not a directory.
[ENOENT]	The named directory does not exist.
[EACCES]	Search permission is denied for any component of the path name.
[EFAULT]	*Path* points outside the allocated address space of the process.
[ELOOP]	A path name lookup involved too many symbolic links.
[ENAMETOOLONG]	The length of *path* exceeds *{PATH_MAX}*, or a pathname component is longer than *{NAME_MAX}*.

SEE ALSO

chroot(2), getwd(3C).

DIAGNOSTICS

Upon successful completion, a value of 0 is returned. Otherwise, a value of −1 is returned and *errno* is set to indicate the error.

NAME

chmod, fchmod – change mode of file

SYNOPSIS

#include <sys/types.h>
#include <sys/stat.h>

int chmod (const char *path, mode_t mode);

int fchmod (int fd, mode_t mode);

DESCRIPTION

The file whose name is given by *path* or referenced by the descriptor *fd* has the access permission portion of the named file's mode changed according to the bit pattern contained in *mode*.

Access permission bits are interpreted as follows:

ISUID	04000	Set user ID on execution.
S_ISGID	020#0	Set group ID on execution if # is **7**, **5**, **3**, or **1**. Enable mandatory file/record locking if # is **6**, **4**, **2**, or **0**
S_ISGID	02000	Set group inheritance (directories only).
S_ISVTX	01000	Sticky bit (see discussion below).
S_IREAD	00400	Read by owner.
S_IWRITE	00200	Write by owner.
S_IEXEC	00100	Execute (search if a directory) by owner.
	00070	Read, write, execute (search) by group.
	00007	Read, write, execute (search) by others.

The effective user ID of the process must match the owner of the file or be super-user to change the mode of a file.

If the effective user ID of the process is not super-user and the effective group ID of the process does not match the group ID of the file, mode bit 02000 (set group ID on execution) is cleared.

The sticky bit 01000 is so named for historical reasons. It currently has no effect on any regular file. If a directory is writable and has the sticky bit set, files within that directory can be removed only if one or more of the following is true [see *unlink*(2)]:

the user owns the file
the user owns the directory
the file is writable by the user
the user is the super-user

If the effective user ID of the process is not super-user, the sticky bit is

cleared for any non-directory argument.

If the mode bit 02000 (set group ID on execution) is set and the mode bit 00010 (execute or search by group) is not set, mandatory file/record locking will exist on a regular file. This may effect future calls to *open*(2), *creat*(2), *read*(2), and *write*(2) on this file.

If the mode bit 02000 is set on a directory, then any files created in that directory will take on the group ID of the directory rather than the group ID of the calling process. *mount*(1M) may be used to enable this feature regardless of the mode of the directory.

chmod will fail and the file mode will be unchanged if one or more of the following are true:

[ENOTDIR]	A component of the path prefix is not a directory.
[ENOENT]	The named file does not exist.
[EACCES]	Search permission is denied on a component of the path prefix.
[EPERM]	The effective user ID does not match the owner of the file and the effective user ID is not super-user.
[EROFS]	The named file resides on a read-only file system.
[EFAULT]	*Path* points outside the allocated address space of the process.
[ELOOP]	A path name lookup involved too many symbolic links.
[ENAMETOOLONG]	The length of *path* exceeds *{PATH_MAX}*, or a path-name component is longer than *{NAME_MAX}*.

fchmod will fail if:

[EBADF]	The descriptor is not valid.
[EINVAL]	*Fd* refers to a socket not a file.
[EROFS]	The file resides on a read only file system.
[EPERM]	The effective user ID does not match the owner of the file and the effective user ID is not super-user.

SEE ALSO

chown(2), creat(2), fchown(2), fcntl(2), mknod(2), open(2), read(2), write(2), fstab(4).
chmod(1), mount(1M) in the *User's Reference Manual.*

DIAGNOSTICS

Upon successful completion, a value of 0 is returned. Otherwise, a value of −1 is returned and *errno* is set to indicate the error.

BUGS

Due to the overloading of various permission bits, the silent turning off of the set group ID on execution bit under the above mentioned circumstances may in fact have disabled mandatory file/record locking (for files) or group inheritance (for directories). The only way to determine if *chmod* really worked in these cases is to *stat(2)* the file.

NAME

chown, fchown – change owner and group of a file (System V and 4.3BSD)

C SYNOPSIS

SysV:

```
#include <sys/types.h>
#include <unistd.h>

int chown (const char *path, uid_t owner, gid_t group);

int fchown (int fd, uid_t owner, gid_t group);
```

BSD:

Links with the BSD version automatically:

```
int BSDchown (const char *path, uid_t owner, gid_t group);

int BSDfchown (int fd, uid_t owner, gid_t group);
```

Links with the BSD version if *–lbsd* is specified during link phase:

```
int chown (const char *path, uid_t owner, gid_t group);

int fchown (int fd, uid_t owner, gid_t group);
```

DESCRIPTION

Path points to a path name naming a file, and *fd* refers to the file descriptor associated with a file. The owner ID and group ID of the named file are set to the numeric values contained in *owner* and *group* respectively.

Only processes with effective user ID equal to the file owner or super-user may change the ownership of a file.

If *chown* is invoked by other than the super-user, the set-user-ID and set-group-ID bits of the file mode, 04000 and 02000 respectively, will be cleared. Note that this has the side-effect of disabling mandatory file/record locking.

The only difference between the System V and 4.3BSD versions is that the 4.3BSD versions allow either the owner or group ID to be left unchanged by specifying it as a –1.

chown will fail and the owner and group of the named file will remain unchanged if one or more of the following are true:

[ENOTDIR]	A component of the path prefix is not a directory.
[ENOENT]	The named file does not exist.
[EACCES]	Search permission is denied on a component of the path prefix.

[EPERM] The effective user ID does not match the owner of the file and the effective user ID is not super-user.

[EROFS] The named file resides on a read-only file system.

[EFAULT] *Path* points outside the allocated address space of the process.

[ELOOP] A path name lookup involved too many symbolic links.

[ENAMETOOLONG] The length of *path* exceeds *{PATH_MAX}*, or a path-name component is longer than *{NAME_MAX}*.

fchown will fail if:

[EBADF] *Fd* does not refer to a valid descriptor.

[EINVAL] *Fd* refers to a socket, not a file.

SEE ALSO

chmod(2), fchmod(2).
chown(1) in the *User's Reference Manual.*

DIAGNOSTICS

Upon successful completion, a value of 0 is returned. Otherwise, a value of −1 is returned and *errno* is set to indicate the error.

NAME

chroot – change root directory

C SYNOPSIS

int chroot (path)
char *path;

DESCRIPTION

Path points to a path name naming a directory. *chroot* causes the named directory to become the root directory, the starting point for path searches for path names beginning with /. The user's working directory is unaffected by the *chroot* system call.

The effective user ID of the process must be super-user to change the root directory.

The .. entry in the root directory is interpreted to mean the root directory itself. Thus, .. cannot be used to access files outside the subtree rooted at the root directory.

chroot will fail and the root directory will remain unchanged if one or more of the following are true:

[ENOTDIR] Any component of the path name is not a directory.

[ENOENT] The named directory does not exist.

[EPERM] The effective user ID is not super-user.

[EFAULT] *Path* points outside the allocated address space of the process.

[EINTR] A signal was caught during the *chroot* system call.

[ENOLINK] *Path* points to a remote machine and the link to that machine is no longer active.

[EMULTIHOP] Components of *path* require hopping to multiple remote machines.

[ELOOP] A path name lookup involved too many symbolic links.

[ENAMETOOLONG] A component of a path name exceeded 255 characters, or an entire path name exceeded 1023 characters.

SEE ALSO

chdir(2).

DIAGNOSTICS

Upon successful completion, a value of 0 is returned. Otherwise, a value of −1 is returned and *errno* is set to indicate the error.

NAME

close – close a file descriptor

C SYNOPSIS

#include <unistd.h>

int close (int fildes);

DESCRIPTION

Fildes is a file descriptor obtained from a *creat*, *open*, *dup*, *fcntl*, or *pipe* system call. *close* closes the file descriptor indicated by *fildes*. All outstanding record locks owned by the process (on the file indicated by *fildes*) are removed.

If a STREAMS [see *intro*(2)] file is closed, and the calling process had previously registered to receive a SIGPOLL signal [see *signal*(2) and *sigset*(2)] for events associated with that file [see I_SETSIG in *streamio*(7)], the calling process will be unregistered for events associated with the file. The last *close* for a *stream* causes the *stream* associated with *fildes* to be dismantled. If O_NDELAY is not set and there have been no signals posted for the *stream*, *close* waits up to 15 seconds, for each module and driver, for any output to drain before dismantling the *stream*. If the O_NDELAY flag is set or if there are any pending signals, *close* does not wait for output to drain, and dismantles the *stream* immediately.

The named file is closed unless one or more of the following are true:

[EBADF] *Fildes* is not a valid open file descriptor.

[EINTR] A signal was caught during the *close* system call.

SEE ALSO

creat(2), dup(2), exec(2), fcntl(2), intro(2), open(2), pipe(2), signal(2), sigset(2).
streamio(7) in the *System Administrator's Reference Manual.*

DIAGNOSTICS

Upon successful completion, a value of 0 is returned. Otherwise, a value of −1 is returned and *errno* is set to indicate the error.

NAME

connect – initiate a connection on a socket

SYNOPSIS

#include <sys/types.h>
#include <sys/socket.h>

connect(s, name, namelen)
int s;
struct sockaddr *name;
int namelen;

DESCRIPTION

The parameter *s* is a socket. If it is of type SOCK_DGRAM, then this call specifies the peer with which the socket is to be associated; this address is that to which datagrams are to be sent, and the only address from which datagrams are to be received. If the socket is of type SOCK_STREAM, then this call attempts to make a connection to another socket. The other socket is specified by *name*, which is an address in the communications space of the socket. Each communications space interprets the *name* parameter in its own way. Generally, stream sockets may successfully *connect* only once; datagram sockets may use *connect* multiple times to change their association. Datagram sockets may dissolve the association by connecting to an invalid address, such as a zero-filled address.

RETURN VALUE

If the connection or binding succeeds, then 0 is returned. Otherwise a –1 is returned, and a more specific error code is stored in *errno*.

ERRORS

The call fails if:

[EBADF]	*S* is not a valid descriptor.
[ENOTSOCK]	*S* is a descriptor for a file, not a socket.
[EADDRNOTAVAIL]	The specified address is not available on this machine.
[EAFNOSUPPORT]	Addresses in the specified address family cannot be used with this socket.
[EISCONN]	The socket is already connected.
[ETIMEDOUT]	Connection establishment timed out without establishing a connection.
[ECONNREFUSED]	The attempt to connect was forcefully rejected.

[ENETUNREACH]	The network isn't reachable from this host.
[EADDRINUSE]	The address is already in use.
[EFAULT]	The *name* parameter specifies an area outside the process address space.
[EINPROGRESS]	The socket is non-blocking and the connection cannot be completed immediately. It is possible to *select*(2) for completion by selecting the socket for writing.
[EALREADY]	The socket is non-blocking and a previous connection attempt has not yet been completed.

SEE ALSO

accept(2), select(2), socket(2)

NAME

creat – create a new file or rewrite an existing one

C SYNOPSIS

```
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>

int creat (const char *path, mode_t mode);
```

DESCRIPTION

creat creates a new ordinary file or prepares to rewrite an existing file named by the path name pointed to by *path*.

If the file exists, the length is truncated to 0 and the mode and owner are unchanged. Otherwise, the file's owner ID is set to the effective user ID, of the process the group ID is set to the effective group ID, of the process or to the group ID of the directory in which the file is being created. This is determined as follows:

> If the underlying filesystem was mounted with the BSD file creation semantics flag [see *fstab*(4)] or the **S_ISGID** bit is set [see *chmod*(2)] on the parent directory, then the group ID of the new file is set to the group ID of the parent directory, otherwise it is set to the effective group ID of the calling process.

The low-order 12 bits of the file mode are set to the value of *mode* modified as follows:

> All bits set in the process's file mode creation mask are cleared [see *umask*(2)].
>
> The "sticky bit" of the mode is cleared [see *chmod*(2)].

Upon successful completion, a write-only file descriptor is returned and the file is open for writing, even if the mode does not permit writing. A new file may be created with a mode that forbids writing.

The file pointer used to mark the current position within the file is set to the beginning of the file.

The new file descriptor is set to remain open across *execve*(2) system calls [see *fcntl*(2)].

There is a system enforced limit on the number of open file descriptors per process *{OPEN_MAX}*, whose value is returned by the *getdtablesize*(2) function.

creat fails if one or more of the following are true:

[ENOTDIR]	A component of the path prefix is not a directory.
[ENOENT]	A component of the path prefix does not exist.
[EACCES]	Search permission is denied on a component of the path prefix.
[ENOENT]	The path name is null.
[EACCES]	The file does not exist and the directory in which the file is to be created does not permit writing.
[EROFS]	The named file resides or would reside on a read-only file system.
[ETXTBSY]	The file is a pure procedure (shared text) file that is being executed.
[EACCES]	The file exists and write permission is denied.
[EISDIR]	The named file is an existing directory.
[EMFILE]	The system imposed limit for open file descriptors per process *{OPEN_MAX}* has already been reached.
[EFAULT]	*Path* points outside the allocated address space of the process.
[ENFILE]	The system file table has exceeded *{NFILE_MAX}* concurrently open files.
[EAGAIN]	The file exists, mandatory file/record locking is set, and there are outstanding record locks on the file [see *chmod*(2)].
[ENOSPC]	The file system is out of inodes.
[ENAMETOOLONG]	The length of *path* exceeds *{PATH_MAX}*, or a path-name component is longer than *{NAME_MAX}*.
[EOPNOTSUPP]	An attempt was made to open a socket (not currently supported).

SEE ALSO

chmod(2), close(2), dup(2), fcntl(2), lseek(2), open(2), read(2), umask(2), write(2), fstab(4).

DIAGNOSTICS

Upon successful completion, a non-negative integer, namely the file descriptor, is returned. Otherwise, a value of −1 is returned and *errno* is set to indicate the error.

NAME

dup – duplicate an open file descriptor

C SYNOPSIS

#include <unistd.h>

int dup (int fildes);

DESCRIPTION

Fildes is a descriptor obtained from a *creat*, *open*, *dup*, *fcntl*, *pipe*, *socket*, or *socketpair* system call. *dup* returns a new descriptor having the following in common with the original:

Refers to same object as the original descriptor.

Same file pointer (i.e., both file descriptors share one file pointer).

Same access mode (read, write or read/write).

Same descriptor status flags (i.e., both descriptors share the same status flags).

Shares any file locks.

The new descriptor is set to remain open across *exec* system calls [see *fcntl*(2)].

The descriptor returned is the lowest one available.

dup will fail if one or more of the following are true:

[EBADF] *Fildes* is not a valid open file descriptor.

[EMFILE] The system imposed limit for open file descriptors per process *{OPEN_MAX}* has already been reached.

SEE ALSO

close(2), creat(2), exec(2), fcntl(2), intro(2), open(2), pipe(2), lockf(3C).

DIAGNOSTICS

Upon successful completion a non-negative integer, namely the file descriptor, is returned. Otherwise, a value of −1 is returned and *errno* is set to indicate the error.

NAME

exec: execl, execv, execle, execve, execlp, execvp – execute a file

SYNOPSIS

```
#include <unistd.h>

int execl (path, arg0, arg1, ..., argn, (char *)0)
const char *path, *arg0, *arg1, ..., *argn;

int execv (path, argv)
const char *path, *argv[ ];

int execle (path, arg0, arg1, ..., argn, (char *)0, envp)
const char *path, *arg0, *arg1, ..., *argn, *envp[ ];

int execve (path, argv, envp)
const char *path, *argv[ ], *envp[ ];

int execlp (file, arg0, arg1, ..., argn, (char *)0)
const char *file, *arg0, *arg1, ..., *argn;

int execvp (file, argv)
const char *file, *argv[ ];
```

DESCRIPTION

exec in all its forms transforms the calling process into a new process. The new process is constructed from an ordinary, executable file called the *new process file*. This file is either an executable object file, or a file of data for an interpreter.

An interpreter file begins with a line of the form "#! *interpreter*". The interpreter string may be a maximum of 64 characters long including the leading "#!". When an interpreter file is *execve*'d, the system *execve*'s the specified *interpreter*, giving it the name of the originally exec'd file as an argument, shifting over the rest of the original arguments. This file consists of a header [see *a.out*(4)], a text segment, and a data segment. The data segment contains an initialized portion and an uninitialized portion (bss). There can be no return from a successful *exec* because the calling process is overlaid by the new process.

When a C program is executed, it is called as follows:

```
main (argc, argv, envp)
int argc;
char **argv, **envp;
```

where *argc* is the argument count, *argv* is an array of character pointers to the arguments themselves, and *envp* is an array of character pointers to the environment strings. As indicated, *argc* is conventionally at least one and the first member of the array points to a string containing the name of the

file.

Path points to a path name that identifies the new process file.

File points to the new process file. The path prefix for this file is obtained by a search of the directories passed as the *environment* line "PATH =" [see *environ*(5)]. The environment is supplied by the shell [see *sh*(1)].

Arg0, *arg1*, ..., *argn* are pointers to null-terminated character strings. These strings constitute the argument list available to the new process. By convention, at least *arg0* must be present and point to a string that is the same as *path* (or its last component).

Argv is an array of character pointers to null-terminated strings. These strings constitute the argument list available to the new process. By convention, *argv* must have at least one member, and it must point to a string that is the same as *path* (or its last component). *Argv* is terminated by a null pointer.

Envp is an array of character pointers to null-terminated strings. These strings constitute the environment for the new process. *Envp* is terminated by a null pointer. For *execl* and *execv*, the C run-time start-off routine places a pointer to the environment of the calling process in the global cell:

extern char **environ;

and it is used to pass the environment of the calling process to the new process.

execlp and *execvp* are called with the same arguments as *execl* and *execv*, resp., but duplicate the shell's actions in searching for an executable file in a list of directories. The directory list is obtained from the environment.

File descriptors open in the calling process remain open in the new process, except for those whose close-on-exec flag is set; see *fcntl*(2). For those file descriptors that remain open, the file pointer is unchanged.

Signals set to terminate the calling process will be set to terminate the new process. Signals set to be ignored by the calling process will be set to be ignored by the new process. Signals set to be caught by the calling process will be set to terminate new process; see *signal*(2).

For signals set by *sigset*(2), *sigaction*(2), or *sigvec*(3B), *exec* will ensure that the new process has the same system signal action for each signal type whose action is SIG_DFL, SIG_IGN, or SIG_HOLD as the calling process. However, if the action is to catch the signal, then the action will be reset to SIG_DFL, and any pending signal for this type will be held. All signal masks associated with handlers are cleared.

If the set-user-ID mode bit of the new process file is set [see *chmod*(2)], *exec* sets the effective user ID of the new process to the owner ID of the new process file. Similarly, if the set-group-ID mode bit of the new process file is set, the effective group ID of the new process is set to the group ID of the new process file. The real user ID, real group ID, and supplementary group IDs of the new process remain the same as those of the calling process. The effective user ID and effective group ID are saved (as the *saved set-user-ID* and the *saved set-group-ID*) for use by *setuid*(2).

The shared memory segments attached to the calling process will not be attached to the new process [see *shmop*(2)].

If the process is a member of a share group, it is removed from that share group [see *sproc*(2)].

Profiling is disabled for the new process; see *profil*(2).

Ability to access graphics is disabled.

The new process also inherits the following attributes from the calling process:

nice value [see *nice*(2)]
process ID
parent process ID
process group ID
session membership
real user and group IDs
supplementary access groups [see *getgroups*(2)]
semadj values [see *semop*(2)]
tty group ID [see *exit*(2) and *signal*(2)]
trace flag [see *ptrace*(2) request 0]
time left until an alarm clock signal [see *alarm*(2)]
interval timers [see *getitimer*(2)]
current working directory
root directory
file mode creation mask [see *umask*(2)]
file size limit [see *ulimit*(2)]
resource limits [see *getrlimit*(2)]
process signal mask [see *sigprocmask*(2)]
pending signals [see *sigpending*(2)]
utime, *stime*, *cutime*, and *cstime* [see *times*(2)]
file-locks [see *fcntl*(2) and *lockf*(3C)]

exec will fail and return to the calling process if one or more of the following are true:

[ENOENT]	One or more components of the new process path name of the file do not exist.
[ENOTDIR]	A component of the new process path of the file prefix is not a directory.
[EACCES]	Search permission is denied for a directory listed in the new process file's path prefix.
[EACCES]	The new process file is not an ordinary file.
[EACCES]	The new process file mode denies execution permission.
[ENOEXEC]	The exec is not an *execlp* or *execvp* , and the new process file has the appropriate access permission but an invalid magic number in its header.
[ENOEXEC]	The new process file has badly formed or missing file header, section header, or optional header.
[ENOEXEC]	The requested virtual addresses are not available.
[ETXTBSY]	The new process file is a pure procedure (shared text) file that is currently open for writing by some process.
[ENOMEM]	The new process requires more virtual space than is allowed either by the system-imposed maximum or the process imposed maximum *{PROCSIZE_MAX}* [see *getrlimit*(2) and *intro*(2)].
[E2BIG]	The number of bytes in the new process's argument list is greater than the system-imposed limit *{ARG_MAX}* [see *sysconf*(2), *intro*(2), and **limits.h**].
[EFAULT]	*Path*, *argv*, or *envp* point to an illegal address.
[EAGAIN]	Not enough memory.
[EPERM]	A non-superuser attempts to execute a setuid or setgid shell script with a uid or gid which is different than the user's effective uid/gid, and the configured value for *nosuidshells* is non-zero (the default) [see *intro*(2) and *lboot(1M)]*.
[ELIBACC]	Required shared library does not have execute permission.
[ELIBMAX]	The required number of shared libraries exceeds the system imposed maximum *{SHLIB_MAX}* [see *intro*(2)].

[ELIBEXEC] Trying to *exec*(2) a shared library directly.

[ENAMETOOLONG] The length of the *path* or *file* arguments, or an element of the environment variable **PATH** prefixed to a file, exceeds *{PATH_MAX}*, or a pathname component is longer than *{NAME_MAX}*.

SEE ALSO

lboot(1M), intro(2), alarm(2), exit(2), fcntl(2), fork(2), getgroups(2), lockf(3C), nice(2), pcreate(3P), ptrace(2), semop(2), sigaction(2), signal(2), sigpending(2), sigprocmask(2), sigset(2), signal(3B), sigvec(3B), sproc(2), sysconf(2), times(2), ulimit(2), umask(2), a.out(4), environ(5).
sh(1) in the *User's Reference Manual.*

DIAGNOSTICS

If *exec* returns to the calling process an error has occurred; the return value will be −1 and *errno* will be set to indicate the error.

NAME

exit, _exit – terminate process

C SYNOPSIS

#include <stdlib.h>

void exit (int status);

#include <unistd.h>

void _exit (int status);

DESCRIPTION

exit terminates the calling process with the following consequences:

All of the file descriptors open in the calling process are closed. If the process is sharing file descriptors via an *sproc*, other members of the share group do NOT have their file descriptors closed.

If the parent process of the calling process is executing a *wait*, it is notified of the calling process's termination and the low order eight bits (i.e., bits 0377) of *status* are made available to it [see *wait*(2)].

If the parent process of the calling process is not executing a *wait*, and the parent process is not ignoring **SIGCLD**, the calling process is transformed into a zombie process. A *zombie process* is a process that only occupies a slot in the process table. It has no other space allocated either in user or kernel space. The process table slot that it occupies contains the time accounting information [see **<sys/proc.h>**] to be used by *times*.

The parent process ID of all of the calling processes' existing child processes and zombie processes is set to 1. This means the initialization process [see *intro*(2)] inherits each of these processes.

If the process belongs to a share group, it is removed from that group. Its stack segment is deallocated and removed from the share group's virtual space. All other virtual space that was shared with the share group is left untouched. If the *prctl* (**PR_SETEXITSIG**) option has been enabled for the share group, than the specified signal is sent to all remaining share group members.

Each attached shared memory segment is detached and the value of **shm_nattach** in the data structure associated with its shared memory identifier is decremented by 1.

For each semaphore for which the calling process has set a semadj value [see *semop*(2)], that semadj value is added to the semval of the specified semaphore.

If the process has a process, text, or data lock, an *unlock* is performed [see *plock*(2)]. If the process has any pages locked, they are unlocked [see *mpin*(2)].

An accounting record is written on the accounting file if the system's accounting routine is enabled [see *acct*(2)].

If the calling process is a process group leader (its process ID matches its process group ID), and it became a process group leader by invoking the *setpgrp*(2) function, and it is a terminal group leader (has an associated controlling terminal), then the **SIGHUP** signal is sent to each process that has a process group ID equal to that of the calling process.

If the calling process is a process group leader (its process ID matches its process group ID), and it became a process group leader by invoking the *setpgid*(2) function, and it is a session leader [see *setsid*(2)], and it has an associated controlling terminal, then the **SIGHUP** signal is sent to each process in the foreground process group of the controlling terminal belonging to calling process.

If the calling process is a process group leader (its process ID matches its process group ID), and it became a process group leader by invoking the *setpgrp*(2) function, no signal is sent.

In all cases, if the calling process is a process group leader and has an associated controlling terminal, the controlling terminal is disassociated from the process allowing it to be acquired by another process group leader.

Any mapped files are closed and any written pages flushed to disk.

A death of child signal is sent to the parent.

The C function *exit* causes all file streams to be closed unless one has done an *sproc* which causes the file streams to simply be flushed. The function *_exit* circumvents all cleanup.

SEE ALSO

acct(2), intro(2), mmap(2), mpin(2), plock(2), prctl(2), semop(2), setpgid(2), setpgrp(2), signal(2), sigset(2), sigaction(2), sigprocmask(2), sigvec(3B), sigblock(3B), sigsetmask(3B), sproc(2), wait(2).

WARNING

See *WARNING* in *signal*(2).

DIAGNOSTICS

None. There can be no return from an *exit* system call.

NAME

fchdir – change working directory, given an open file descriptor

C SYNOPSIS

int fchdir (fd)
int fd;

DESCRIPTION

Fd is a file descriptor of a directory that the calling process has opened. *fchdir* causes the directory with that file descriptor to become the process's current working directory.

fchdir will fail and the current working directory will be unchanged if one or more of the following are true:

[EBADF]	*Fd* is not a valid open file descriptor.
[ENOTDIR]	*Fd* does not refer to a directory.
[EACCES]	Search permission is denied for the directory.

SEE ALSO

chdir(2).

DIAGNOSTICS

Upon successful completion, a value of 0 is returned. Otherwise, a value of −1 is returned and *errno* is set to indicate the error.

NAME

fcntl – file and descriptor control

C SYNOPSIS

#include <unistd.h>
#include <fcntl.h>

int fcntl (int fildes, int cmd, ...);

DESCRIPTION

fcntl provides for control over open descriptors. *Fildes* is an open descriptor obtained from a *creat*, *open*, *dup*, *fcntl*, *pipe*, *socket*, or *socketpair* system call.

The commands available are:

F_DUPFD Return a new descriptor as follows:

Lowest numbered available descriptor greater than or equal to the third argument, *arg*, taken as an integer of type int.

Refers to the same object as the original descriptor.

Same file pointer as the original file (i.e., both file descriptors share one file pointer).

Same access mode (read, write or read/write).

Same descriptor status flags (i.e., both descriptors share the same status flags).

Shares any file locks.

The close-on-exec flag (**FD_CLOEXEC**) associated with the new descriptor is cleared to keep the file open across calls to the *exec*(2) family of functions.

F_GETFD Get the file descriptor flags associated with the descriptor *fildes*. If the **FD_CLOEXEC** flag is **0** the descriptor will remain open across *exec*, otherwise the descriptor will be closed upon execution of *exec*.

F_SETFD Set the file descriptor flags for *fildes*. Currently the only flag implemented is **FD_CLOEXEC**. Note: this flag is a per-process and per-descriptor flag; setting or clearing it for a particular descriptor will not affect the flag on descriptors copied from it by a *dup*(2) or **F_DUPFD** operation, nor will it affect the flag on other processes instances of that descriptor.

F_GETFL Get *file* status flags and file access modes. The file access modes may be extracted from the return value using the mask **O_ACCMODE**.

F_SETFL Set *file* status flags to the third argument, *arg*, taken as type int. Only the following flags can be set [see *fcntl*(5)]: **FAPPEND**, **FSYNC**, **FNDELAY**, **FNONBLOCK**, and **FASYNC**. **FAPPEND** is equivalent to **O_APPEND**; **FSYNC** is equivalent to **O_SYNC**; **FNDELAY** is equivalent to **O_NDELAY**; and **FNONBLOCK** is equivalent to **O_NONBLOCK**. **FASYNC** is equivalent to calling *ioctl* with the **FIOASYNC** command. This enables the **SIGIO** facilities and is currently supported only on sockets.

Since the descriptor status flags are shared with descriptors copied from a given descriptor by a *dup*(2) or **F_DUPFD** operation, and by other processes instances of that descriptor a **F_SETFL** operation will affect those other descriptors and other instances of the given descriptors as well. For example, setting or clearing the **FNDELAY** flag will logically cause an **FIONBIO** *ioctl*(2) to be performed on the object referred to by that descriptor. Thus all descriptors referring to that object will be affected.

Flags not understood for a particular descriptor are silently ignored.

F_GETLK Get the first lock which blocks the lock description given by the variable of type *struct flock* pointed to by *arg*. The information retrieved overwrites the information passed to *fcntl* in the *flock* structure. If no lock is found that would prevent this lock from being created, then the structure is passed back unchanged except for the lock type which will be set to **F_UNLCK**.

F_SETLK Set or clear a file segment lock according to the variable of type *struct flock* pointed to by *arg* [see *fcntl*(5)]. The *cmd* **F_SETLK** is used to establish read (F_RDLCK) and write (F_WRLCK) locks, as well as remove either type of lock (F_UNLCK). If a read or write lock cannot be set *fcntl* will return immediately with an error value of –1.

F_SETLKW This *cmd* is the same as **F_SETLK** except that if a read or write lock is blocked by other locks, the process will sleep until the segment is free to be locked.

F_CHKFL This flag is used internally by **F_SETFL** to check the legality of file flag changes.

F_GETOWN Used by sockets: get the process ID or process group currently receiving **SIGIO** and **SIGURG** signals; process groups are returned as negative values.

F_SETOWN Used by sockets: set the process or process group to receive SIGIO and SIGURG signals; process groups are specified by supplying *arg* as negative, otherwise *arg* is interpreted as a process ID.

A read lock prevents any process from write locking the protected area. More than one read lock may exist for a given segment of a file at a given time. The file descriptor on which a read lock is being placed must have been opened with read access.

A write lock prevents any process from read locking or write locking the protected area. Only one write lock may exist for a given segment of a file at a given time. The file descriptor on which a write lock is being placed must have been opened with write access.

The structure *flock* describes the type (*l_type*), starting offset (*l_whence*), relative offset (*l_start*), size (*l_len*), process id (*l_pid*), and RFS system id (*l_sysid*) of the segment of the file to be affected. The process id and system id fields are used only with the **F_GETLK** *cmd* to return the values for a blocking lock. Locks may start and extend beyond the current end of a file, but may not be negative relative to the beginning of the file. A lock may be set to always extend to the end of file by setting *l_len* to zero (0). If such a lock also has *l_whence* and *l_start* set to zero (0), the whole file will be locked. Changing or unlocking a segment from the middle of a larger locked segment leaves two smaller segments for either end. Locking a segment that is already locked by the calling process causes the old lock type to be removed and the new lock type to take effect. All locks associated with a file for a given process are removed when a file descriptor for that file is closed by that process or the process holding that file descriptor terminates. Locks are not inherited by a child process in a *fork*(2) system call.

When mandatory file and record locking is active on a file, [see *chmod*(2)], *read* and *write* system calls issued on the file will be affected by the record locks in effect.

fcntl will fail if one or more of the following are true:

[EBADF] *Fildes* is not a valid open file descriptor.

[EINVAL] *Cmd* is F_DUPFD. *arg* is either negative, or greater than or equal to the maximum number of open file descriptors allowed each user [see *getdtablesize*(2)].

[EMFILE] *Cmd* is F_DUPFD and *{OPEN_MAX}* file descriptors are currently in use by this process, or no file descriptors greater than or equal to *arg* are available.

[EINVAL] *Cmd* is F_GETLK, F_SETLK, or SETLKW and *arg* or the data it points to is not valid.

[EAGAIN] *Cmd* is F_SETLK the type of lock (*l_type*) is a read (F_RDLCK) lock and the segment of a file to be locked is already write locked by another process or the type is a write (F_WRLCK) lock and the segment of a file to be locked is already read or write locked by another process.

[ENOLCK] *Cmd* is F_SETLK or F_SETLKW, the type of lock is a read or write lock, and there are no more record locks available (too many file segments locked) because the system maximum *{FLOCK_MAX}* [see *intro*(2)], has been exceeded.

[EINTR] *Cmd* is F_SETLKW and a signal interrupted the process while it was waiting for the lock to be granted.

[EDEADLK] *Cmd* is F_SETLKW, the lock is blocked by some lock from another process, and putting the calling-process to sleep, waiting for that lock to become free, would cause a deadlock.

[EFAULT] *Cmd* is F_SETLK, *arg* points outside the program address space.

[ESRCH] *Cmd* is F_SETOWN and no process can be found corresponding to that specified by *arg*.

SEE ALSO

close(2), creat(2), dup(2), exec(2), fork(2), getdtablesize(2), intro(2), open(2), pipe(2), fcntl(5).

DIAGNOSTICS

Upon successful completion, the value returned depends on *cmd* as follows:

F_DUPFD	A new file descriptor.
F_GETFD	Value of flag (only the low-order bit is defined).
F_SETFD	Value other than −1.
F_GETFL	Value of file flags.
F_SETFL	Value other than −1.
F_GETLK	Value other than −1.
F_SETLK	Value other than −1.
F_SETLKW	Value other than −1.
F_GETOWN	*Pid* of socket owner.
F_SETOWN	Value other than −1.

Otherwise, a value of −1 is returned and *errno* is set to indicate the error.

NAME

fork – create a new process

C SYNOPSIS

#include <sys/types.h>
#include <unistd.h>

pid_t fork (void);

DESCRIPTION

fork causes creation of a new process. The new process (child process) is an exact copy of the calling process (parent process). This means the child process inherits the following attributes from the parent process:

environment
close-on-exec flag [see *exec*(2)]
signal handling settings (i.e., **SIG_DFL**, **SIG_IGN**, **SIG_HOLD**, function addresses and signal masks)
set-user-ID mode bit
set-group-ID mode bit
profiling on/off status
debugger trailing status
nice value [see *nice*(2)]
all attached shared memory segments [see *shmop*(2)]
all mapped files [see *mmap*(2)]
non-degrading priority [see *schedctl*(2)]
process group ID
tty group ID [see *exit*(2)]
current working directory
root directory
file mode creation mask [see *umask*(2)]
file size limit [see *ulimit*(2)]

The child process differs from the parent process in the following ways:

The child process has a unique process ID.

The child process has a different parent process ID (i.e., the process ID of the parent process).

The child process has its own copy of the parent's file descriptors. Each of the child's file descriptors shares a common file pointer with the corresponding file descriptor of the parent.

File locks previously set by the parent are not inherited by the child [see *fcntl*(2)].

All semadj values are cleared [see *semop*(2)].

Process locks, text locks and data locks are not inherited by the child [see *plock*(2)].

The set of signals pending to the parent is not inherited by the child.

Page locks are not inherited [see *mpin*(2)].

The child process's *utime*, *stime*, *cutime*, and *cstime* are set to 0. The time left until an alarm clock signal is reset to 0.

The time left until an itimer signal is reset to 0.

The child will not inherit the ability to make graphics calls. The child must establish itself as a graphics process by invoking the *winopen*(3G) (or *ginit*(3G)) call. Otherwise the child process may receive a segmentation fault upon attempting to make a graphics call.

The share mask is set to 0 [see *sproc*(2)].

fork will fail and no child process will be created if one or more of the following are true:

[EAGAIN] The system-imposed limit on the total number of processes under execution, *{NPROC}* [see *intro*(2)], would be exceeded.

[EAGAIN] The system-imposed limit on the total number of processes under execution by a single user *{CHILD_MAX}* [see *intro*(2)], would be exceeded.

[EAGAIN] Amount of system memory required is temporarily unavailable.

SEE ALSO

exec(2), intro(2), mmap(2), nice(2), pcreate(3C), plock(2), ptrace(2), schedctl(2), semop(2), shmop(2), signal(2), sigset(2), sigaction(2), signal(3B), sigvec(3B), sproc(2), times(2), ulimit(2), umask(2), wait(2).

DIAGNOSTICS

Upon successful completion, *fork* returns a value of 0 to the child process and returns the process ID of the child process to the parent process. Otherwise, a value of −1 is returned to the parent process, no child process is created, and *errno* is set to indicate the error.

NAME

fsync – synchronize a file's in-core state with that on disk

SYNOPSIS

int fsync (fd)
int fd;

DESCRIPTION

fsync causes all modified data and attributes of *fd* to be moved to a permanent storage device. This normally results in all in-core modified copies of buffers for the associated file to be written to a disk. *fsync* should be used by programs that require a file to be in a known state; for example in building a simple transaction facility.

fsync will fail if one or more of the following are true:

[EBADF] *Fd* is not a valid file descriptor.

SEE ALSO

sync(2)

DIAGNOSTICS

Upon successful completion, *fsync* returns 0; else -1 and *errno* is set to the proper error number.

NAME

getdents – read directory entries and put in a file system independent format

C SYNOPSIS

```
#include <sys/dirent.h>

int getdents (fildes, buf, nbyte)
int fildes;
char *buf;
unsigned nbyte;
```

DESCRIPTION

Fildes is a file descriptor obtained from an *open*(2) or *dup*(2) system call.

getdents attempts to read *nbyte* bytes from the directory associated with *fildes* and to format them as file system independent directory entries in the buffer pointed to by *buf*. Since the file system independent directory entries are of variable length, in most cases the actual number of bytes returned will be strictly less than *nbyte*.

The file system independent directory entry is specified by the *dirent* structure. For a description of this see *dirent*(4).

On devices capable of seeking, *getdents* starts at a position in the file given by the file pointer associated with *fildes*. Upon return from *getdents*, the file pointer is incremented to point to the next directory entry.

This system call was developed in order to implement the *readdir*(3C) routine [for a description see *directory*(3C)], and should not be used for other purposes.

getdents will fail if one or more of the following are true:

[EBADF]	*Fildes* is not a valid file descriptor open for reading.
[EFAULT]	*Buf* points outside the allocated address space.
[EINVAL]	*nbyte* is not large enough for one directory entry.
[ENOENT]	The current file pointer for the directory is not located at a valid entry.
[ENOLINK]	*Fildes* points to a remote machine and the link to that machine is no longer active.
[ENOTDIR]	*Fildes* is not a directory.
[EIO]	An I/O error occurred while accessing the file system.

SEE ALSO

directory(3C), dirent(4).

DIAGNOSTICS

Upon successful completion a non-negative integer is returned indicating the number of bytes actually read. A value of 0 indicates the end of the directory has been reached. If the system call failed, a −1 is returned and *errno* is set to indicate the error.

NAME

getdomainname, setdomainname – get/set name of current domain

SYNOPSIS

getdomainname(name, namelen)
char *name;
int namelen;

setdomainname(name, namelen)
char *name;
int namelen;

DESCRIPTION

Getdomainname returns the name of the domain for the current processor, as previously set by *setdomainname*. The parameter *namelen* specifies the size of the *name* array. The returned name is null-terminated unless insufficient space is provided.

Setdomainname sets the domain of the host machine to be *name*, which has length *namelen*. This call is restricted to the super-user and is normally used only when the system is bootstrapped.

The purpose of domains is to enable two distinct networks that may have host names in common to merge. Each network would be distinguished by having a different domain name. At the current time, only the Yellow Pages service makes use of domains.

RETURN VALUE

If the call succeeds a value of 0 is returned. If the call fails, then a value of −1 is returned and an error code is placed in the global location *errno*.

ERRORS

The following errors may be returned by these calls:

[EFAULT] The *name* parameter gave an invalid address.

[EPERM] The caller was not the super-user. This error only applies to *setdomainname*.

BUGS

Domain names are limited to 64 characters.

NAME

getgroups – get group access list

SYNOPSIS

#include <sys/param.h>

int ngrps;

POSIX:

ngrps = getgroups(int setlen, gid_t *gidset);

BSD:

ngrps = getgroups(int setlen, int *gidset);

To use the BSD versions of *setgroups*, *getgroups*, or *initgroups*, you must either

1) explicitly invoke them as *BSDsetgroups*, *BSDgetgroups*, or *BSDinitgroups*, or

2) link with the libbsd.a library:

```
cc -o prog prog.c -lbsd
```

DESCRIPTION

getgroups retrieves the current group access list of the user process and stores it in the array *gidset*. The parameter *setlen* indicates the number of entries that may be placed in *gidset*. *getgroups* returns the actual number of groups returned in *gidset*. No more than **NGROUPS**, as defined in <*sys/param.h*>, will ever be returned.

As a special case, if the *setlen* parameter is zero, *getgroups* returns the number of supplemental group IDs associated with the calling process without modifying the array pointed to by the *gidset* argument.

RETURN VALUE

A successful call returns the number of groups in the group set. A value of −1 indicates that an error occurred, and the error code is stored in the global variable *errno*.

ERRORS

The possible errors for *getgroups* are:

[EINVAL]	The argument *setlen* is smaller than the number of groups in the group set.
[EFAULT]	The argument *gidset* specifies an invalid address.

SEE ALSO

multgrps(1), setgroups(2), initgroups(3)

CAVEATS

The POSIX and 4.3BSD versions differ in the types of their *gidset* parameter.

NAME

gethostid, sethostid – get/set unique identifier of current host

C SYNOPSIS

```
hostid = gethostid()
long hostid;

sethostid(hostid)
long hostid;
```

DESCRIPTION

Sethostid establishes a 32-bit identifier for the current processor that is intended to be unique among all UNIX systems in existence. This is normally a DARPA Internet address for the local machine. This call is allowed only to the super-user and is normally performed at boot time.

Gethostid returns the 32-bit identifier for the current processor.

SEE ALSO

hostid(1), gethostname(2)

BUGS

32 bits for the identifier is too small.

NAME

gethostname, sethostname – get/set name of current host

C SYNOPSIS

gethostname(name, namelen)
char *name;
int namelen;

sethostname(name, namelen)
char *name;
int namelen;

DESCRIPTION

Gethostname returns the standard host name for the current processor, as previously set by *sethostname*. The parameter *namelen* specifies the size of the *name* array. The returned name is null-terminated unless insufficient space is provided.

Sethostname sets the name of the host machine to be *name*, which has length *namelen*. This call is restricted to the super-user and is normally used only when the system is bootstrapped.

RETURN VALUE

If the call succeeds a value of 0 is returned. If the call fails, then a value of −1 is returned and an error code is placed in the global location *errno*.

ERRORS

The following errors may be returned by these calls:

[EFAULT]	The *name* or *namelen* parameter gave an invalid address.
[EPERM]	The caller tried to set the hostname and was not the super-user.
[EINVAL]	The *namelen* parameter was too large.

SEE ALSO

gethostid(2)

BUGS

Host names are limited to MAXHOSTNAMELEN (from <*sys/param.h*>) characters, currently 64.

NAME

getitimer, setitimer – get/set value of interval timer

SYNOPSIS

```
#include <sys/time.h>

#define ITIMER_REAL      0      /* real time intervals */
#define ITIMER_VIRTUAL   1      /* virtual time intervals */
#define ITIMER_PROF      2      /* user and system virtual time

int getitimer(int which, struct itimerval *value);

int setitimer(int which, struct itimerval *value,
              struct itimerval *ovalue);
```

DESCRIPTION

The system provides each process with three interval timers, defined in **<sys/time.h>**. The *getitimer* call returns the current value for the timer specified in *which*, while the *setitimer* call sets the value of a timer (optionally returning the previous value of the timer).

A timer value is defined by the *itimerval* structure:

```
struct itimerval {
        struct  timeval it_interval;        /* timer interval */
        struct  timeval it_value; /* current value */
};
```

If *it_value* is non-zero, it indicates the time to the next timer expiration and not the time the timer was set originally. If *it_interval* is non-zero, it specifies a value to be used in reloading *it_value* when the timer expires. Setting *it_value* to 0 disables a timer. Setting *it_interval* to 0 causes a timer to be disabled after its next expiration (assuming *it_value* is non-zero).

Time values smaller than the resolution of the timer clock are rounded up to this resolution. The default timer resolution is 10 milliseconds unless the operating system is configured with the variable *fastimer_enable* set. The variable *fastimer_enable* can be found in the file **master.d/kernel** [see *lboot*(1M)]. The command *ftimer*(1) can be used to enable and disable fast itimers at run time. If fast itimers are enabled then the resolution depends on the fast clock resolution [see *ftimer*(1))]. Fast itimer interrupts are always handled by the processor that handles the system clock function [see *ftimer*(1), and *mpadmin*(1)]. Fast itimer requests can reduce the system performance by 6-8%. To not bog down the system, itimer requests with resolutions that are higher than one second or that are a multiple of 10 millisecond will not invoke the fast itimer facility.

NAME

getmsg – get next message off a stream

SYNOPSIS

#include <stropts.h>

int getmsg(fd, ctlptr, dataptr, flags)
int fd;
struct strbuf *ctlptr;
struct strbuf *dataptr;
int *flags;

DESCRIPTION

getmsg retrieves the contents of a message [see *intro*(2)] located at the *stream head* read queue from a STREAMS file, and places the contents into user specified buffer(s). The message must contain either a data part, a control part or both. The data and control parts of the message are placed into separate buffers, as described below. The semantics of each part is defined by the STREAMS module that generated the message.

Fd specifies a file descriptor referencing an open *stream*. *Ctlptr* and *dataptr* each point to a *strbuf* structure which contains the following members:

```
int maxlen;     /* maximum buffer length */
int len;        /* length of data        */
char *buf;      /* ptr to buffer         */
```

where *buf* points to a buffer in which the data or control information is to be placed, and *maxlen* indicates the maximum number of bytes this buffer can hold. On return, *len* contains the number of bytes of data or control information actually received, or is 0 if there is a zero-length control or data part, or is -1 if no data or control information is present in the message. *Flags* may be set to the values 0 or RS_HIPRI and is used as described below.

Ctlptr is used to hold the control part from the message and *dataptr* is used to hold the data part from the message. If *ctlptr* (or *dataptr*) is NULL or the *maxlen* field is -1, the control (or data) part of the message is not processed and is left on the *stream head* read queue and *len* is set to -1. If the *maxlen* field is set to 0 and there is a zero-length control (or data) part, that zero-length part is removed from the read queue and *len* is set to 0. If the *maxlen* field is set to 0 and there are more than zero bytes of control (or data) information, that information is left on the read queue and *len* is set to 0. If the *maxlen* field in *ctlptr* or *dataptr* is less than, respectively, the control or data part of the message, *maxlen* bytes are retrieved. In this case, the remainder of the message is left on the *stream head* read queue and a non-zero return value is provided, as described below under *DIAGNOSTICS*. If information is retrieved from a *priority* message, *flags* is set to RS_HIPRI on return.

By default, *getmsg* processes the first priority or non-priority message available on the *stream head* read queue. However, a user may choose to retrieve only priority messages by setting *flags* to RS_HIPRI. In this case, *getmsg* will only process the next message if it is a priority message.

If O_NDELAY has not been set, *getmsg* blocks until a message, of the type(s) specified by *flags* (priority or either), is available on the *stream head* read queue. If O_NDELAY has been set and a message of the specified type(s) is not present on the read queue, *getmsg* fails and sets *errno* to EAGAIN.

If a hangup occurs on the *stream* from which messages are to be retrieved, *getmsg* will continue to operate normally, as described above, until the *stream head* read queue is empty. Thereafter, it will return 0 in the *len* fields of *ctlptr* and *dataptr*.

getmsg fails if one or more of the following are true:

[EAGAIN] The O_NDELAY flag is set, and no messages are available.

[EBADF] *Fd* is not a valid file descriptor open for reading.

[EBADMSG] Queued message to be read is not valid for *getmsg*.

[EFAULT] *Ctlptr*, *dataptr*, or *flags* points to a location outside the allocated address space.

[EINTR] A signal was caught during the *getmsg* system call.

[EINVAL] An illegal value was specified in *flags*, or the *stream* referenced by *fd* is linked under a multiplexor.

[ENOSTR] A *stream* is not associated with *fd*.

A *getmsg* can also fail if a STREAMS error message had been received at the *stream head* before the call to *getmsg*. The error returned is the value contained in the STREAMS error message.

SEE ALSO

intro(2), read(2), poll(2), putmsg(2), write(2).
STREAMS Primer
STREAMS Programmer's Guide

DIAGNOSTICS

Upon successful completion, a non-negative value is returned. A value of 0 indicates that a full message was read successfully. A return value of MORECTL indicates that more control information is waiting for retrieval. A return value of MOREDATA indicates that more data is waiting for retrieval. A return value of MORECTL|MOREDATA indicates that both types of information remain. Subsequent *getmsg* calls will retrieve the

remainder of the message.

NAME

getpagesize – get system page size

SYNOPSIS

pagesize = getpagesize()
int pagesize;

DESCRIPTION

Getpagesize returns the number of bytes in a page. Page granularity is the granularity of many of the memory management calls.

The page size is a *system* page size and may not be the same as the underlying hardware page size.

SEE ALSO

sbrk(2)

NAME

getpeername – get name of connected peer

SYNOPSIS

#include <sys/types.h>
#include <sys/socket.h>

getpeername(s, name, namelen)
int s;
struct sockaddr *name;
int *namelen;

DESCRIPTION

Getpeername returns the name of the peer connected to socket *s*. The *namelen* parameter should be initialized to indicate the amount of space pointed to by *name*. On return it contains the actual size of the name returned (in bytes). The name is truncated if the buffer provided is too small.

DIAGNOSTICS

A 0 is returned if the call succeeds, –1 if it fails.

ERRORS

The call succeeds unless:

[EBADF]	The argument *s* is not a valid descriptor.
[ENOTSOCK]	The argument *s* is a file, not a socket.
[ENOTCONN]	The socket is not connected.
[ENOBUFS]	Insufficient resources were available in the system to perform the operation.
[EFAULT]	The *name* parameter points to memory not in a valid part of the process address space.

SEE ALSO

accept(2), bind(2), socket(2), getsockname(2)

NAME

getpid, getpgrp, getppid – get process, process group, and parent process IDs

C SYNOPSIS

```
#include <unistd.h>
#include <sys/types.h>

pid_t getpid (void);

pid_t getpgrp (void);

pid_t getppid (void);
```

DESCRIPTION

getpid returns the process ID of the calling process.

getpgrp returns the process group ID of the calling process.

getppid returns the parent process ID of the calling process.

SEE ALSO

exec(2), fork(2), intro(2), setpgrp(2), signal(2).

NAME

getpriority, setpriority – get/set program scheduling priority

SYNOPSIS

#include <sys/resource.h>

prio = getpriority(which, who)
int prio, which, who;

setpriority(which, who, prio)
int which, who, prio;

DESCRIPTION

The scheduling priority of the process, process group, or user, as indicated by *which* and *who* is obtained with the *getpriority* call and set with the *setpriority* call. *Which* is one of **PRIO_PROCESS**, **PRIO_PGRP**, or **PRIO_USER**, and *who* is interpreted relative to *which* (a process identifier for **PRIO_PROCESS**, process group identifier for **PRIO_PGRP**, and a user ID for **PRIO_USER**). A zero value of *who* denotes the current process, process group, or user. *Prio* is a value in the range –20 to 20. The default priority is 0; lower priorities cause more favorable scheduling.

The *getpriority* call returns the highest priority (lowest numerical value) enjoyed by any of the specified processes. The *setpriority* call sets the priorities of all of the specified processes to the specified value. Only the super-user may lower priorities.

getpriority and *setpriority* may return one of the following errors:

[ESRCH] No process was located using the *which* and *who* values specified.

[EINVAL] *Which* was not one of PRIO_PROCESS, PRIO_PGRP, or PRIO_USER.

In addition to the errors indicated above, *setpriority* may fail with one of the following errors returned:

[EPERM] A process was located, but neither its effective nor real user ID matched the effective user ID of the caller.

[EACCES] A non super-user attempted to lower a process priority.

SEE ALSO

renice(1M), fork(2), nice(1), schedctl(2).

DIAGNOSTICS

Since *getpriority* can legitimately return the value –1, it is necessary to clear the external variable *errno* prior to the call, then check it afterward to determine if a –1 is an error or a legitimate value. The *setpriority* call returns 0 if there is no error, or –1 if there is.

NAME

getrlimit, setrlimit – control maximum system resource consumption

SYNOPSIS

#include <sys/resource.h>

getrlimit(resource, rlp)
int resource;
struct rlimit *rlp;

setrlimit(resource, rlp)
int resource;
struct rlimit *rlp;

DESCRIPTION

Limits on the consumption of system resources by the current process and each process it creates may be obtained with the *getrlimit* call, and set with the *setrlimit* call.

The *resource* parameter is one of the following:

RLIMIT_CPU the maximum amount of cpu time (in seconds) to be used by each process.

RLIMIT_FSIZE the largest size, in bytes, of any single file that may be created.

RLIMIT_DATA the maximum size, in bytes, of the data segment for a process; this defines how far a program may extend its break with the *sbrk*(2) system call.

RLIMIT_STACK the maximum size, in bytes, of the stack segment for a process; this defines how far a program's stack segment may be extended. Stack extension is performed automatically by the system.

RLIMIT_CORE the largest size, in bytes, of a *core* file that may be created.

RLIMIT_RSS the maximum size, in bytes, to which a process's resident set size may grow. This imposes a limit on the amount of physical memory to be given to a process; if memory is tight, the system will prefer to take memory from processes that are exceeding their declared resident set size.

A resource limit is specified as a soft limit and a hard limit. When a soft limit is exceeded a process may receive a signal (for example, if the cpu time is exceeded), but it will be allowed to continue execution until it reaches the hard limit (or modifies its resource limit). The *rlimit* structure is used to specify the hard and soft limits on a resource,

```
struct rlimit {
        int     rlim_cur;       /* current (soft) limit */
        int     rlim_max;       /* hard limit */
};
```

Only the super-user may raise the maximum limits. Other users may only alter *rlim_cur* within the range from 0 to *rlim_max* or (irreversibly) lower *rlim_max*.

An ''infinite'' value for a limit is defined as **RLIM_INFINITY** (0x7fffffff).

Because this information is stored in the per-process information, this system call must be executed directly by the shell if it is to affect all future processes created by the shell; *limit* is thus a built-in command to *sh*(1) and *csh*(1).

The system refuses to extend the data or stack space when the limits would be exceeded in the normal way: a *break* call fails if the data space limit is reached. When the stack limit is reached, the process receives a segmentation fault (**SIGSEGV**). Since delivery of the signal would require the kernel to extend the stack, this signal can not be caught.

A file I/O operation that would create a file that is too large will cause a signal **SIGXFSZ** to be generated; this normally terminates the process, but may be caught. When the soft cpu time limit is exceeded, a signal **SIGXCPU** is sent to the offending process.

getrlimit or *setrlimit* will fail if one or more of the following are true:

[EFAULT] The address specified for *rlp* is invalid.

[EPERM] The limit specified to *setrlimit* would have raised the maximum limit value, and the caller is not the super-user.

[EINVAL] The *resource* specified is not a valid resource, or one of the values of the resource specified to *setrlimit* is less than zero.

SEE ALSO

sh(1), csh(1).

DIAGNOSTICS

A 0 return value indicates that the call succeeded, changing or returning the resource limit. A return value of −1 indicates that an error occurred, and an error code is stored in the global location *errno*.

NAME

getsockname – get socket name

SYNOPSIS

```
#include <sys/types.h>
#include <sys/socket.h>

getsockname(s, name, namelen)
int s;
struct sockaddr *name;
int *namelen;
```

DESCRIPTION

Getsockname returns the current *name* for the specified socket. The *namelen* parameter should be initialized to indicate the amount of space pointed to by *name*. On return it contains the actual size of the name returned (in bytes).

DIAGNOSTICS

A 0 is returned if the call succeeds, –1 if it fails.

ERRORS

The call succeeds unless:

[EBADF]	The argument *s* is not a valid descriptor.
[ENOTSOCK]	The argument *s* is a file, not a socket.
[ENOBUFS]	Insufficient resources were available in the system to perform the operation.
[EFAULT]	The *name* parameter points to memory not in a valid part of the process address space.

SEE ALSO

bind(2), socket(2)

NAME

getsockopt, setsockopt – get and set options on sockets

C SYNOPSIS

#include <sys/types.h>
#include <sys/socket.h>

int
getsockopt(int s, int level, int optname, void *optval, int *optlen)

int
setsockopt(int s, int level, int optname, const void *optval, int optlen)

DESCRIPTION

Getsockopt and *setsockopt* manipulate *options* associated with a socket. Options may exist at multiple protocol levels; they are always present at the uppermost ''socket'' level.

When manipulating socket options the level at which the option resides and the name of the option must be specified. To manipulate options at the ''socket'' level, *level* is specified as SOL_SOCKET. To manipulate options at any other level the protocol number of the appropriate protocol controlling the option is supplied. For example, to indicate that an option is to be interpreted by the TCP protocol, *level* should be set to the protocol number of TCP; see *getprotoent*(3N).

The parameters *optval* and *optlen* are used to access option values for *setsockopt*. For *getsockopt* they identify a buffer in which the value for the requested option(s) are to be returned. For *getsockopt*, *optlen* is a value-result parameter, initially containing the size of the buffer pointed to by *optval*, and modified on return to indicate the actual size of the value returned. If no option value is to be supplied or returned, *optval* may be supplied as 0.

Optname and any specified options are passed uninterpreted to the appropriate protocol module for interpretation. The include file <*sys/socket.h*> contains definitions for ''socket'' level options, described below. Options at other protocol levels vary in format and name; consult the appropriate entries in section (4P).

Most socket-level options take an *int* parameter for *optval*. For *setsockopt*, the parameter should non-zero to enable a boolean option, or zero if the option is to be disabled. SO_LINGER uses a *struct linger* parameter, defined in <*sys/socket.h*>, which specifies the desired state of the option and the linger interval (see below).

The following options are recognized at the socket level. Except as noted, each may be examined with *getsockopt* and set with *setsockopt*.

SO_DEBUG	toggle recording of debugging information
SO_REUSEADDR	toggle local address reuse
SO_KEEPALIVE	toggle keep connections alive
SO_DONTROUTE	toggle routing bypass for outgoing messages
SO_LINGER	linger on close if data present
SO_BROADCAST	toggle permission to transmit broadcast messages
SO_OOBINLINE	toggle reception of out-of-band data in band
SO_SNDBUF	set buffer size for output
SO_RCVBUF	set buffer size for input
SO_TYPE	get the type of the socket (get only)
SO_ERROR	get and clear error on the socket (get only)

SO_DEBUG enables debugging in the underlying protocol modules. SO_REUSEADDR indicates that the rules used in validating addresses supplied in a *bind*(2) call should allow reuse of local addresses. SO_KEEPALIVE enables the periodic transmission of messages on a connected socket. Should the connected party fail to respond to these messages, the connection is considered broken and processes using the socket are notified via a SIGPIPE signal. SO_DONTROUTE indicates that outgoing messages should bypass the standard routing facilities. Instead, messages are directed to the appropriate network interface according to the network portion of the destination address.

SO_LINGER controls the action taken when unsent messages are queued on socket and a *close*(2) is performed. If the socket promises reliable delivery of data and SO_LINGER is set, the system will block the process on the *close* attempt until it is able to transmit the data or until it decides it is unable to deliver the information (a timeout period, termed the linger interval, is specified in the *setsockopt* call when SO_LINGER is requested). If SO_LINGER is disabled and a *close* is issued, the system will process the close in a manner that allows the process to continue as quickly as possible.

The option SO_BROADCAST requests permission to send broadcast datagrams on the socket. Broadcast was a privileged operation in earlier versions of the system. With protocols that support out-of-band data, the SO_OOBINLINE option requests that out-of-band data be placed in the normal data input queue as received; it will then be accessible with *recv* or *read* calls without the MSG_OOB flag. SO_SNDBUF and SO_RCVBUF are options to adjust the normal buffer sizes allocated for output and input buffers, respectively. The buffer size may be increased for high-volume connections, or may be decreased to limit the possible backlog of incoming

data. The system places an absolute limit on these values. Finally, SO_TYPE and SO_ERROR are options used only with *setsockopt*.

SO_TYPE returns the type of the socket, such as SOCK_STREAM; it is useful for servers that inherit sockets on startup. SO_ERROR returns any pending error on the socket and clears the error status. It may be used to check for asynchronous errors on connected datagram sockets or for other asynchronous errors.

RETURN VALUE

A 0 is returned if the call succeeds, −1 if it fails.

ERRORS

The call succeeds unless:

[EBADF]	The argument *s* is not a valid descriptor.
[ENOTSOCK]	The argument *s* is a file, not a socket.
[ENOPROTOOPT]	The option is unknown at the level indicated.
[EFAULT]	The address pointed to by *optval* is not in a valid part of the process address space. For *getsockopt*, this error may also be returned if *optlen* is not in a valid part of the process address space.

SEE ALSO

ioctl(2), socket(2), getprotoent(3N)

BUGS

Several of the socket options should be handled at lower levels of the system.

NAME

getuid, geteuid, getgid, getegid – get real user, effective user, real group, and effective group IDs

C SYNOPSIS

```
#include <unistd.h>
#include <sys/types.h>

uid_t getuid (void);
uid_t geteuid (void);


gid_t getgid (void);
gid_t getegid (void);
```

DESCRIPTION

getuid returns the real user ID of the calling process.

geteuid returns the effective user ID of the calling process.

getgid returns the real group ID of the calling process.

getegid returns the effective group ID of the calling process.

SEE ALSO

intro(2), setuid(2).

NAME

ioctl – control device

C SYNOPSIS

int ioctl (int fildes, int request, ...);

DESCRIPTION

ioctl performs a variety of control functions on devices and STREAMS. For non-STREAMS files, the functions performed by this call are *device-specific* control functions. The *request* and optional third argument are passed to the file designated by *fildes* and are interpreted by the device driver. For a given device, the *requests* that are understood are documented in the section 7 manual page for that device. This control is infrequently used on non-STREAMS devices, with the basic input/output functions performed through the *read*(2) and *write*(2) system calls.

For STREAMS files, specific functions are performed by the *ioctl* call as described in *streamio*(7).

Fildes is an open file descriptor that refers to a device. *Request* selects the control function to be performed and will depend on the device being addressed. The optional third argument represents additional information that is needed by this specific device to perform the requested function. The data type of the third argument depends upon the particular control request, but it is either an integer or a pointer to a device-specific data structure.

In addition to device-specific and STREAMS functions, generic functions are provided by more than one device driver, for example, the general terminal interface [see *termio*(7)].

ioctl will fail for any type of file if one or more of the following are true:

[EACCES] Future error.

[EBADF] *Fildes* is not a valid open file descriptor.

[ENOTTY] *Fildes* is not associated with a device driver that accepts control functions.

[EINTR] A signal was caught during the *ioctl* system call.

ioctl will also fail if the device driver detects an error. In this case, the error is passed through *ioctl* without change to the caller. A particular driver might not have all of the following error cases. Other requests to device drivers will fail if one or more of the following are true:

[EFAULT] *Request* requires a data transfer to or from a buffer pointed to by the third argument but some part of the buffer is outside the process's allocated space.

[EINVAL] *Request* or the third argument is not valid for this device.

[EIO] Some physical I/O error has occurred.

[ENXIO] The *request* and the third argument are valid for this device driver, but the service requested can not be performed on this particular subdevice.

STREAMS errors are described in *streamio*(7).

SEE ALSO

streamio(7), termio(7) in the *System Administrator's Reference Manual.*

DIAGNOSTICS

Upon successful completion, the value returned depends upon the device control function, but must be a non-negative integer. Otherwise, a value of −1 is returned and *errno* is set to indicate the error.

NAME

kill – send a signal to a process or a group of processes

C SYNOPSIS

#include <sys/types.h>
#include <signal.h>

int kill (pid_t pid, int sig);

DESCRIPTION

kill sends a signal to a process or a group of processes. The process or group of processes to which the signal is to be sent is specified by *pid*. The signal that is to be sent is specified by *sig* and is either one from the list given in *signal*(2), or 0. If *sig* is 0 (the null signal), error checking is performed but no signal is actually sent. This can be used to check the validity of *pid*.

The real or effective user ID of the sending process must match the real, saved, or effective user ID of the receiving process, unless the effective user ID of the sending process is super-user. An exception to this is the signal SIGCONT, which may be sent to any descendant, or any process in the same session (having the same session ID) as the current process.

The processes with a process ID of 0 and a process ID of 1 are special processes [see *intro*(2)] and will be referred to below as *proc0* and *proc1*, respectively.

If *pid* is greater than zero, *sig* will be sent to the process whose process ID is equal to *pid*. *Pid* may equal 1.

If *pid* is 0, *sig* will be sent to all processes excluding *proc0* and *proc1* whose process group ID is equal to the process group ID of the sender.

If *pid* is –1 and the effective user ID of the sender is not super-user, *sig* will be sent to all processes excluding *proc0* and *proc1* whose real user ID is equal to the effective user ID of the sender.

If *pid* is –1 and the effective user ID of the sender is super-user, *sig* will be sent to all processes excluding *proc0* and *proc1*.

If *pid* is negative but not –1, *sig* will be sent to all processes whose process group ID is equal to the absolute value of *pid*.

kill will fail and no signal will be sent if one or more of the following are true:

[EINVAL] *Sig* is not a valid signal number.

[EINVAL] *Sig* is SIGKILL and *pid* is 1 (proc1).

[ESRCH] No process can be found corresponding to that specified by *pid*.

[ESRCH] The process group was given as 0 but the sending process does not have a process group.

[EPERM] The user ID of the sending process is not super-user, and its real or effective user ID does not match the real, saved, or effective user ID of the receiving process.

SEE ALSO

exec(2), getpid(2), setpgrp(2), setsid(2), signal(2), sigset(2), sigaction(2), sigprocmask(2), sigvec(3B), sigblock(3B), sigsetmask(3B), killpg(3B).
kill(1) in the *User's Reference Manual.*

DIAGNOSTICS

Upon successful completion, a value of 0 is returned. Otherwise, a value of −1 is returned and *errno* is set to indicate the error.

NAME

link – link to a file

C SYNOPSIS

#include <unistd.h>

int link (const char *path1, const char *path2);

DESCRIPTION

Path1 points to a path name naming an existing file. *Path2* points to a path name naming the new directory entry to be created. *link* creates a new link (directory entry) for the existing file.

link will fail and no link will be created if one or more of the following are true:

[ENOTDIR] A component of either path prefix is not a directory.

[ENOENT] A component of either path prefix does not exist.

[EACCES] A component of either path prefix denies search permission.

[ENOENT] The file named by *path1* does not exist.

[EEXIST] The link named by *path2* exists.

[ENAMETOOLONG] The length of the *path1* or *path2* argument exceeds *{PATH_MAX}*, or a pathname component is longer than *{NAME_MAX}*.

[EPERM] The file named by *path1* is a directory and the effective user ID is not super-user.

[EXDEV] The link named by *path2* and the file named by *path1* are on different logical devices (file systems).

[ENOENT] *Path2* points to a null path name.

[EACCES] The requested link requires writing in a directory with a mode that denies write permission.

[EROFS] The requested link requires writing in a directory on a read-only file system.

[EFAULT] *Path* points outside the allocated address space of the process.

[EMLINK] The maximum number of links to a file would be exceeded.

SEE ALSO

unlink(2).

DIAGNOSTICS

Upon successful completion, a value of 0 is returned. Otherwise, a value of −1 is returned and *errno* is set to indicate the error.

NAME

listen – listen for connections on a socket

SYNOPSIS

listen(s, backlog)
int s, backlog;

DESCRIPTION

To accept connections, a socket is first created with *socket*(2), a willingness to accept incoming connections and a queue limit for incoming connections are specified with *listen*(2), and then the connections are accepted with *accept*(2). The *listen* call applies only to sockets of type SOCK_STREAM.

The *backlog* parameter defines the maximum length the queue of pending connections may grow to. If a connection request arrives with the queue full the client may receive an error with an indication of ECONNREFUSED, or, if the underlying protocol supports retransmission, the request may be ignored so that retries may succeed.

RETURN VALUE

A 0 return value indicates success; –1 indicates an error.

ERRORS

The call fails if:

[EBADF]	The argument *s* is not a valid descriptor.
[ENOTSOCK]	The argument *s* is not a socket.
[EOPNOTSUPP]	The socket is not of a type that supports the operation *listen*.

SEE ALSO

accept(2), connect(2), socket(2)

BUGS

The *backlog* is currently limited (silently) to 5.

NAME

lseek – move read/write file pointer (System V and 4.3BSD)

C SYNOPSIS

SysV:

#include <sys/types.h>
#include <unistd.h>

off_t lseek (int fildes, off_t offset, int whence);

BSD:

#include <sys/file.h>

off_t lseek (int fildes, off_t offset, int whence);

DESCRIPTION

Fildes is a file descriptor returned from a *creat*, *open*, *dup*, or *fcntl* system call. *lseek* sets the file pointer associated with *fildes* as follows:

If *whence* is **SEEK_SET (L_SET)**, the pointer is set to *offset* bytes.

If *whence* is **SEEK_CUR (L_INCR)**, the pointer is set to its current location plus *offset*.

If *whence* is **SEEK_END (L_XTND)**, the pointer is set to the size of the file plus *offset*.

Upon successful completion, the resulting pointer location, as measured in bytes from the beginning of the file, is returned. Note that if *fildes* is a remote file descriptor and *offset* is negative, *lseek* will return the file pointer even if it is negative.

lseek allows the file offset to be set beyond the end of existing data in the file. If data is later written at that point, subsequent reads of the data in the gap return bytes with the value zero until data is actually written into the gap.

lseek will fail and the file pointer will remain unchanged if one or more of the following are true:

[EBADF] *Fildes* is not an open file descriptor.

[ESPIPE] *Fildes* is associated with a pipe, socket, or fifo.

[EINVAL and SIGSYS signal]
Whence is not a proper value.

[EINVAL] *Fildes* is not a remote file descriptor, and the resulting file pointer would be negative.

Some devices are incapable of seeking. The value of the file pointer associated with such a device is undefined.

SEE ALSO

creat(2), dup(2), fcntl(2), open(2).

DIAGNOSTICS

Upon successful completion, a non-negative integer indicating the file pointer value is returned. Otherwise, a value of −1 is returned and *errno* is set to indicate the error.

NAME

madvise – give advise about handling memory

SYNOPSIS

#include <sys/mman.h>

int madvise(void *addr, int len, int behavior);

DESCRIPTION

madvise provides the system advice about the process's expected use of its address space from *addr* to *addr* + *len*.

The following types of *behavior* are currently recognized by the system:

MADV_DONTNEED

informs the system that the address range from *addr* to *addr* + *len* will likely not be referenced in the near future. The memory to which the indicated addresses are mapped will be the first to be reclaimed when memory is needed by the system.

madvise will fail if:

[ENOMEM] Addresses in the range (*addr*, *addr* + *len*) are outside the valid range for the address space of a process.

[EINVAL] *behavior* is not recognized by the system.

SEE ALSO

mmap(2), msync(2).

DIAGNOSTICS

A 0 value is returned on success; a -1 value indicates an error.

BUGS

MADV_DONTNEED advise is currently only heeded for memory mapped via explicit *mmap(2)* system calls.

NAME

mkdir – make a directory

C SYNOPSIS

#include <sys/types.h>
#include <sys/stat.h>

int mkdir (const char *path, mode_t mode);

DESCRIPTION

The routine *mkdir* creates a new directory with the name *path*. The mode of the new directory is initialized from the *mode*. The protection part of the *mode* argument is modified by the process's mode mask [see *umask*(2)].

The directory's owner ID is set to the process's effective user ID. The directory's group ID is set to the process's effective group ID or the group ID of the directory in which the directory is being created. This is determined as follows:

> If the underlying filesystem was mounted with the BSD file creation semantics flag [see *fstab*(4)] or the **S_ISGID** bit is set [see *chmod*(2)] on the parent directory, then the group ID of the new file is set to the group ID of the parent directory, otherwise it is set to the effective group ID of the calling process.

The newly created directory is empty with the possible exception of entries for "." and "..".

mkdir will fail and no directory will be created if one or more of the following are true:

[ENOTDIR]	A component of the path prefix is not a directory.
[ENOENT]	A component of the path prefix does not exist.
[ENAMETOOLONG]	The length of the *path* argument exceeds *{PATH_MAX}*, or a pathname component is longer than *{NAME_MAX}*.
[EACCES]	Either a component of the path prefix denies search permission or write permission is denied on the parent directory of the directory to be created.
[ENOENT]	The path is longer than the maximum allowed.
[EEXIST]	The named file already exists.
[EROFS]	The path prefix resides on a read-only file system.

[EFAULT]	*Path* points outside the allocated address space of the process.
[EMLINK]	The maximum number of links to the parent directory would exceed *{LINK_MAX}*.
[ENOSPC]	No space is available.
[EIO]	An I/O error has occurred while accessing the file system.

SEE ALSO

mkdir(1), chmod(2), mknod(2), unlink(2), fstab(4).

DIAGNOSTICS

Upon successful completion, a value of 0 is returned. Otherwise, a value of -1 is returned, and *errno* is set to indicate the error.

NAME

mkfifo – make a FIFO special file

C SYNOPSIS

#include <sys/types.h>
#include <sys/stat.h>

int mkfifo (char *path, mode_t mode);

DESCRIPTION

The *mkfifo* routine creates a new FIFO special file named by the pathname pointed to by *path*. IRIX implements it via the following *mknod* call:

mknod(path, (mode | S_IFIFO), 0)

where S_IFIFO is defined in <*sys/stat.h*>. Refer to *mknod*(2) for details.

SEE ALSO

mknod(2), chmod(2), exec(2), pipe(2), stat(2), umask(2).

DIAGNOSTICS

Upon successful completion, *mkfifo* returns a value of 0. Otherwise, a value of −1 is returned, no FIFO is created, and *errno* is set to indicate the error.

NAME

mknod – make a directory, or a special or ordinary file

C SYNOPSIS

```
#include <sys/types.h>
#include <sys/stat.h>

int mknod (const char *path, mode_t mode, dev_t dev);
```

DESCRIPTION

mknod creates a new file named by the path name pointed to by *path*. The mode of the new file (including file type bits) is initialized from *mode*. The value of the file type bits which are permitted are:

S_IFIFO fifo special
S_IFCHR character special
S_IFBLK block special
S_IFREG ordinary file

All other mode bits are interpreted as described in *chown*(2).

The owner ID of the file is set to the effective user ID of the process. The group ID of the file is set to the effective group ID of the process or the group ID of the directory in which the file is being created. This is determined as follows:

> If the underlying filesystem was mounted with the BSD file creation semantics flag [see *fstab*(4)] or the S_ISGID bit is set [see *chmod*(2)] on the parent directory, then the group ID of the new file is set to the group ID of the parent directory, otherwise it is set to the effective group ID of the calling process.

Values of *mode* other than those above are undefined and should not be used. The low-order 9 bits of *mode* are modified by the process's file mode creation mask: all bits set in the process's file mode creation mask are cleared [see *umask*(2)]. If *mode* indicates a block or character special file, *dev* is a configuration-dependent specification of a character or block I/O device. If *mode* does not indicate a block special or character special device, *dev* is ignored.

mknod may be invoked only by the super-user for file types other than FIFO special.

mknod will fail and the new file will not be created if one or more of the following are true:

[EPERM]	The effective user ID of the process is not super-user.
[ENOTDIR]	A component of the path prefix is not a directory.
[ENOENT]	A component of the path prefix does not exist.
[EROFS]	The directory in which the file is to be created is located on a read-only file system.
[EEXIST]	The named file exists.
[EFAULT]	*Path* points outside the allocated address space of the process.
[ENAMETOOLONG]	The length of the *path* argument exceeds *{PATH_MAX}*, or a pathname component is longer than *{NAME_MAX}*.
[ENOSPC]	No space is available.
[EINVAL]	If you create files of the type fifo special, character special, or block special on an NFS-mounted file system.

SEE ALSO

chmod(2), exec(2), mkdir(2), umask(2), fstab(4).
mkdir(1) in the *User's Reference Manual.*

DIAGNOSTICS

Upon successful completion a value of 0 is returned. Otherwise, a value of −1 is returned and *errno* is set to indicate the error.

NAME

mmap – map pages of memory

SYNOPSIS

#include <sys/types.h>
#include <sys/mman.h>

void *mmap(void *addr, int len, int prot, int flags, int fd, off_t off);
void *paddr;

DESCRIPTION

mmap establishes a mapping between the process's address space at an address *paddr* for *len* bytes to the object represented by *fd* at *off* for *len* bytes. The value of *paddr* is an implementation-dependent function of the parameter *addr* and values of *flags*, further described below. A successful *mmap* call returns *paddr* as its result.

The parameter *prot* determines whether *read*, *execute*, *write*, or some combination of accesses are permitted to the pages being mapped. The following protection options are defined in *<sys/mman.h>*:

PROT_READ Page can be read.

PROT_WRITE Page can be written.

PROT_EXECUTE Page can be executed.

The parameter *flags* provides other information about the handling of the mapped pages. The following options are defined in *<sys/mman.h>*:

MAP_SHARED All write references will change the object.

MAP_PRIVATE Write references will not change the object. The initial write reference will cause a private copy of the page of the file object to be created.

MAP_AUTOGROW If **MAP_AUTOGROW** is specified, the mapped object will be implicitly grown when referenced by a store operation (write) if the store is beyond the current end of the object (or if the store in within the last page of the object's mapping and the object is mapped beyond its end). The object will be grown and zero-filled to the next page boundary [see *getpagesize*(2)] beyond the reference, or to the end of the mapping, whichever is less.
MAP_AUTOGROW requires that the object is mapped with **PROT_WRITE** permission. Read references to mapped pages following the end of a object will result in the delivery of a **SIGSEGV** signal, as will various filesystem conditions on stores. Whenever a

SIGSEGV signal is delivered, the second argument to the signal handler contains a value that indicates the reason for the delivery of the signal; these values are defined in */usr/include/sys/errno.h*.

MAP_LOCAL If the process does an *sproc*(2) each process will receive a private copy of the object's mapping. All subsequent write reference of objects mapped **MAP_PRIVATE** will cause private copies of the object to be created. In addition, the share group processes will be able to independently unmap the object from their address spaces.

MAP_FIXED **MAP_FIXED** informs the system that the value of *paddr* must be *addr*, exactly. The mapping thus established by *mmap* replaces any previous mappings for the process's pages in the range (*addr, addr + len*). The use of **MAP_FIXED** is discouraged, as it may prevent an implementation from making the most effective use of system resources.
When **MAP_FIXED** is not set, the system uses *addr* as a hint in an implementation-defined manner to arrive at *paddr*. The *paddr* so chosen will be an area of the address space which the system deems suitable for a mapping of *len* bytes to the specified object. When the system selects a value for *paddr*, it will never place a mapping at address 0, nor will it replace any extant mapping.

The parameter *fd* is the file descriptor returned from a *creat*, *open* or *dup* system call.

The parameter *off* is constrained to be aligned and sized according to the value returned by *getpagesize*(2). When **MAP_FIXED** is specified, the parameter *addr* must also meet these constraints. The system performs mapping operations over whole pages. Thus, while the parameter *len* need not meet a size or alignment constraint, the system will include in any mapping operation any partial page specified by the range (*paddr, paddr + len*).

It should be noted that the system will always zero-fill any partial pages at the end of an object. Further, the system will never write out any modified portions of the last page of an object which are beyond the end of the mapping. Any reference to pages outside of a mapping will result in the delivery of a SIGSEGV signal.

mmap will fail if:

[EBADF] *Fd* is not open.

[EACCES] *Fd* is not open for read and **PROT_READ** or **PROT_EXECUTE** were specified, or *fd* is not open for write and **PROT_WRITE** was specified for a **MAP_SHARED** type mapping.

[ENXIO] Addresses in the range (*off*, *off* + *len*) are invalid for *fd* and **MAP_AUTOGROW** was not specified.

[EINVAL] The arguments *addr* (if MAP_FIXED was specified) and *off* are not multiples of the page size as returned by *getpagesize*(2).

[EINVAL] Invalid MAP_TYPE field in *flags* (neither MAP_PRIVATE nor MAP_SHARED).

[ENODEV] *Fd* refers to an object for which *mmap* is meaningless, such as a terminal.

[ENOMEM] **MAP_FIXED** was specified, and the range (*addr*, *addr* + *len*) exceeds that allowed for the address space of a process; or the process would exceed its maximum allowable virtual size *{PROCSIZE_MAX}*; or **MAP_FIXED** was not specified and no suitably large virtual address space is available.

SEE ALSO

madvise(2), munmap(2), msync(2), sigaction(2), signal(2), sigset(2), sproc(2), sigvec(3B).

DIAGNOSTICS

A successful *mmap* returns the address at which the mapping was placed (*paddr*). A failing *mmap* returns -1 and sets *errno* to indicate the error.

NAME

mount – mount a file system

C SYNOPSIS

#include <sys/mount.h>

int mount (spec, dir, [mflag, fstyp, [data, datalen]])
const char *spec, *dir;
int mflag, fstyp;
void *data;
int datalen;

DESCRIPTION

mount attaches a file system to a directory. After a successful return, references to directory *dir* will refer to the root directory on the newly mounted file system. *Dir* is a pointer to a pathname of a existent directory. Its old contents are inaccessible while the filesystem is mounted. *Spec* and *dir* are pointers to path names. *Fstyp* is the file system type number. The *sysfs*(2) system call can be used to determine the file system type number. Note that if the MS_FSS flag bit of *mflag* is off, the file system type will default to the root file system type. Only if the bit is on will *fstyp* be used to indicate the file system type.

The low-order bit of *mflag* is used to control write permission on the mounted file system; if **1**, writing is forbidden, otherwise writing is permitted according to individual file accessibility.

mount may be invoked only by the super-user. It is intended for use only by the *mount*(1M) utility.

mount will fail if one or more of the following are true:

[EPERM]	The effective user ID is not super-user.
[ENOENT]	Any of the named files does not exist.
[ENOTDIR]	A component of a path prefix is not a directory.
[ENOTBLK]	*Spec* is not a block special device.
[ENXIO]	The device associated with *spec* does not exist.
[ENOTDIR]	*Dir* is not a directory.
[EFAULT]	*Spec* or *dir* points outside the allocated address space of the process.
[EBUSY]	*Dir* is currently mounted on, is someone's current working directory, or is otherwise busy.

[EBUSY]	The device associated with *spec* is currently mounted.
[EBUSY]	There are no more mount table entries.
[EROFS]	*Spec* is write protected and *mflag* requests write permission.
[ENOSPC]	The file system state in the super-block is not FsOKAY and *mflag* requests write permission.
[EINVAL]	The super block has an invalid magic number or the *fstyp* is invalid or *mflag* is not valid.

SEE ALSO

sysfs(2), umount(2), fs(4).
mount(1M) in the *System Administrator's Reference Manual.*

DIAGNOSTICS

Upon successful completion a value of 0 is returned. Otherwise, a value of −1 is returned and *errno* is set to indicate the error.

NAME

mpin, munpin – lock pages in memory

SYNOPSIS

int mpin (void *addr, unsigned len);

int munpin (void *addr, unsigned len);

DESCRIPTION

mpin reads into memory all pages over the range (*addr, addr + len*), and locks the pages into memory. Associated with each locked page is a counter which is incremented each time the page is locked. The super-user can lock as many pages as it wishes, other users are limited to a configurable per process maximum.

munpin decrements the lock counter associated with the pages over the range (*addr, addr + len*). Pages whose counters are zero are available to be swapped out at the system's discretion.

mpin or *munpin* will fail if one or more of the following are true:

[EINVAL] The addresses specified by *(addr, addr + len)* are not mapped into the user's address space.

[EAGAIN] There was insufficient lockable memory to lock the entire address range *(addr, addr + len)*. This may occur even though the amount requested was less than the system-imposed maximum number of locked pages.

[ENOMEM] The caller was not super-user and the number of pages to be locked exceeded the per process limit *{PLOCK_MAX}*.

[ENOMEM] The total number of pages locked by the caller would exceed the maximum resident size for the process [see *setrlimit*(2)].

SEE ALSO

intro(2), getrlimit(2), plock(2), ulimit(2).

DIAGNOSTICS

Upon successful completion, *mpin* and *munpin* return 0. Otherwise, a value of –1 is returned and *errno* is set to indicate the error.

NAME

msgctl – message control operations

SYNOPSIS

#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>

int msgctl (int msqid, int cmd, ...);

Type of optional third argument:
struct msqid_ds *buf;

DESCRIPTION

msgctl provides a variety of message control operations as specified by *cmd*. The following *cmd*s are available:

IPC_STAT Place the current value of each member of the data structure associated with *msqid* into the structure pointed to by *buf*. The contents of this structure are defined in *intro*(2). {READ}

IPC_SET Set the value of the following members of the data structure associated with *msqid* to the corresponding value found in the structure pointed to by *buf*:

msg_perm.uid
msg_perm.gid
msg_perm.mode /* only low 9 bits */
msg_qbytes

This *cmd* can only be executed by a process that has an effective user ID equal to either that of super user, or to the value of **msg_perm.cuid** or **msg_perm.uid** in the data structure associated with *msqid*. Only super user can raise the value of **msg_qbytes**.

IPC_RMID Remove the message queue identifier specified by *msqid* from the system and destroy the message queue and data structure associated with it. This *cmd* can only be executed by a process that has an effective user ID equal to either that of super user, or to the value of **msg_perm.cuid** or **msg_perm.uid** in the data structure associated with *msqid*.

msgctl will fail if one or more of the following are true:

[EINVAL] *Msqid* is not a valid message queue identifier.

[EINVAL] *Cmd* is not a valid command.

[EACCES] *Cmd* is equal to **IPC_STAT** and {READ} operation permission is denied to the calling process [see *intro*(2)].

[EPERM] *Cmd* is equal to **IPC_RMID** or **IPC_SET**. The effective user ID of the calling process is not equal to that of super user, or to the value of **msg_perm.cuid** or **msg_perm.uid** in the data structure associated with *msqid*.

[EPERM] *Cmd* is equal to **IPC_SET,** an attempt is being made to increase to the value of **msg_qbytes,** and the effective user ID of the calling process is not equal to that of super user.

[EFAULT] *Buf* points to an illegal address.

SEE ALSO

intro(2), msgget(2), msgop(2).

DIAGNOSTICS

Upon successful completion, a value of 0 is returned. Otherwise, a value of −1 is returned and *errno* is set to indicate the error.

NAME

msgget – get message queue

SYNOPSIS

#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>

int msgget (key_t key, int msgflg);

DESCRIPTION

msgget returns the message queue identifier associated with *key*.

A message queue identifier and associated message queue and data structure [see *intro*(2)] are created for *key* if one of the following are true:

Key is equal to **IPC_PRIVATE**.

Key does not already have a message queue identifier associated with it, and (*msgflg* & **IPC_CREAT**) is "true".

Upon creation, the data structure associated with the new message queue identifier is initialized as follows:

Msg_perm.cuid, **msg_perm.uid**, **msg_perm.cgid**, and **msg_perm.gid** are set equal to the effective user ID and effective group ID, respectively, of the calling process.

The low-order 9 bits of **msg_perm.mode** are set equal to the low-order 9 bits of *msgflg*.

Msg_qnum, **msg_lspid**, **msg_lrpid**, **msg_stime**, and **msg_rtime** are set equal to 0.

Msg_ctime is set equal to the current time.

Msg_qbytes is set equal to the system limit.

msgget will fail if one or more of the following are true:

[EACCES] A message queue identifier exists for *key*, but operation permission [see *intro*(2)] as specified by the low-order 9 bits of *msgflg* would not be granted.

[ENOENT] A message queue identifier does not exist for *key* and (*msgflg* & **IPC_CREAT**) is "false".

[ENOSPC] A message queue identifier is to be created but the system-imposed limit on the maximum number of allowed message queue identifiers system wide would be exceeded.

[EEXIST] A message queue identifier exists for *key* but ((*msgflg* & **IPC_CREAT**) & (*msgflg* & **IPC_EXCL**)) is ''true''.

SEE ALSO

intro(2), msgctl(2), msgop(2).

DIAGNOSTICS

Upon successful completion, a non-negative integer, namely a message queue identifier, is returned. Otherwise, a value of −1 is returned and *errno* is set to indicate the error.

NAME

msgop: msgsnd, msgrcv – message operations

SYNOPSIS

#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>

int msgsnd (int msqid, const struct msgbuf *msgp,
int msgsz, int msgflg);

int msgrcv (int msqid, struct msgbuf *msgp,
int msgsz, long msgtyp, int msgflg);

DESCRIPTION

msgsnd is used to send a message to the queue associated with the message queue identifier specified by *msqid*. {WRITE} *Msgp* points to a structure containing the message. This structure is composed of the following members:

```
long    mtype;      /* message type */
char    mtext[];    /* message text */
```

Mtype is a positive integer that can be used by the receiving process for message selection (see *msgrcv* below). *Mtext* is any text of length *msgsz* bytes. *Msgsz* can range from 0 to a system-imposed maximum.

Msgflg specifies the action to be taken if one or more of the following are true:

The number of bytes already on the queue is equal to **msg_qbytes** [see *intro*(2)].

The total number of messages on all queues system-wide is equal to the system-imposed limit.

These actions are as follows:

If (*msgflg* & **IPC_NOWAIT**) is "true", the message will not be sent and the calling process will return immediately.

If (*msgflg* & **IPC_NOWAIT**) is "false", the calling process will suspend execution until one of the following occurs:

The condition responsible for the suspension no longer exists, in which case the message is sent.

Msqid is removed from the system [see *msgctl*(2)]. When this occurs, *errno* is set equal to EIDRM, and a value of –1 is returned.

The calling process receives a signal that is to be caught. In this case the message is not sent and the calling process resumes execution in the manner prescribed in *signal*(2).

msgsnd will fail and no message will be sent if one or more of the following are true:

[EINVAL] *Msqid* is not a valid message queue identifier.

[EACCES] Operation permission is denied to the calling process [see *intro*(2)].

[EINVAL] *Mtype* is less than 1.

[EAGAIN] The message cannot be sent for one of the reasons cited above and (*msgflg* & **IPC_NOWAIT**) is "true"

[EINVAL] *Msgsz* is less than zero or greater than the system-imposed limit.

[EFAULT] *Msgp* points to an illegal address.

Upon successful completion, the following actions are taken with respect to the data structure associated with *msqid* [see intro (2)].

Msg_qnum is incremented by 1.

Msg_lspid is set equal to the process ID of the calling process.

Msg_stime is set equal to the current time.

msgrcv reads a message from the queue associated with the message queue identifier specified by *msqid* and places it in the structure pointed to by *msgp*. {READ} This structure is composed of the following members:

```
long    mtype;      /* message type */
char    mtext[];    /* message text */
```

Mtype is the received message's type as specified by the sending process. *Mtext* is the text of the message. *Msgsz* specifies the size in bytes of *mtext*. The received message is truncated to *msgsz* bytes if it is larger than *msgsz* and (*msgflg* & **MSG_NOERROR**) is "true". The truncated part of the message is lost and no indication of the truncation is given to the calling process.

Msgtyp specifies the type of message requested as follows:

If *msgtyp* is equal to 0, the first message on the queue is received.

If *msgtyp* is greater than 0, the first message of type *msgtyp* is received.

If *msgtyp* is less than 0, the first message of the lowest type that is less than or equal to the absolute value of *msgtyp* is received.

Msgflg specifies the action to be taken if a message of the desired type is not on the queue. These are as follows:

If (*msgflg* & **IPC_NOWAIT**) is "true", the calling process will return immediately with a return value of −1 and *errno* set to ENOMSG.

If (*msgflg* & **IPC_NOWAIT**) is "false", the calling process will suspend execution until one of the following occurs:

A message of the desired type is placed on the queue.

Msqid is removed from the system. When this occurs, *errno* is set equal to EIDRM, and a value of −1 is returned.

The calling process receives a signal that is to be caught. In this case a message is not received and the calling process resumes execution in the manner prescribed in *signal*(2).

msgrcv will fail and no message will be received if one or more of the following are true:

[EINVAL] *Msqid* is not a valid message queue identifier.

[EACCES] Operation permission is denied to the calling process.

[EINVAL] *Msgsz* is less than 0.

[E2BIG] Mtext is greater than *msgsz* and (*msgflg* & **MSG_NOERROR**) is "false".

[ENOMSG] The queue does not contain a message of the desired type and (*msgtyp* & **IPC_NOWAIT**) is "true".

[EFAULT] *Msgp* points to an illegal address.

Upon successful completion, the following actions are taken with respect to the data structure associated with *msqid* [see intro (2)].

Msg_qnum is decremented by 1.

Msg_lrpid is set equal to the process ID of the calling process.

Msg_rtime is set equal to the current time.

SEE ALSO

intro(2), msgctl(2), msgget(2), signal(2).

DIAGNOSTICS

If *msgsnd* or *msgrcv* return due to the receipt of a signal a value of −1 is returned to the calling process and *errno* is set to EINTR. If they return due to removal of *msqid* from the system, a value of −1 is returned and *errno* is set to EIDRM.

Upon successful completion, the return value is as follows:

Msgsnd returns a value of 0.

Msgrcv returns a value equal to the number of bytes actually placed into *mtext*.

Otherwise, a value of −1 is returned and *errno* is set to indicate the error.

NAME

msync – synchronize memory with physical storage

SYNOPSIS

#include <sys/types.h>
#include <sys/mman.h>

int msync(void *addr, int len, int flags);

DESCRIPTION

msync causes all modified copies of pages over the range (*addr, addr + len*) to be written to their permanent storage locations. *msync* optionally invalidates any copies so that further references to the pages will cause the system to obtain them from their permanent storage locations.

The various *flags* are used to control the behavior of *msync*. One or more flags may be specified in a single call. The following values for *flags* are defined in <*sys/mman.h*>:

MS_ASYNC causes *msync* to return immediately once all I/O operations are scheduled; normally, *msync* will not return until all I/O operations are complete.

MS_INVALIDATE causes all cached copies of data from memory objects to be invalidated, requiring them to be re-obtained from the object's permanent storage location upon the next reference.

msync should be used by programs which require a memory object to be in a known state, for example, in building transaction facilities.

msync will fail if:

[EIO] An I/O error occurred while reading from or writing to the file system.

[ENOMEM] Addresses in the range (*addr, addr + len*) are outside the valid range for the address space of a process.

[EBUSY] **MS_INVALIDATE** was specified and one or more of the pages was locked in memory.

SEE ALSO

mmap(2), mpin(2).

DIAGNOSTICS

A 0 value is returned on success. A -1 value indicates an error.

NAME

munmap – unmap pages of memory

SYNOPSIS

#include <sys/types.h>
#include <sys/mman.h>

int munmap(void *addr, int len);

DESCRIPTION

munmap causes the mappings for pages in the range (*addr*, *addr* + *len*) to be removed. Further references to these pages will result in the delivery of a **SIGSEGV** signal to the process. Only pages that have been mapped by *mmap(2)* may be unmapped.

An *mmap* can perform an implicit *munmap*.

Current implementations do not support unmapping portions of memory regions. (*addr*, *addr* + *len*) must describe an entire memory region mapped by a previous *mmap* call.

munmap will fail if:

[EINVAL]	Addresses in the range (*addr*, *addr* + *len*) were not mapped via *mmap*.
[EINVAL]	(*addr*, *addr* + *len*) does not describe an entire memory region previous mapped via *mmap*.
[ENOMEM]	Addresses in the range (*addr*, *addr* + *len*) are outside the valid range for the address space of a process.

SEE ALSO

mmap(2), signal(2).

DIAGNOSTICS

munmap returns 0 on success, -1 on failure.

NAME

nice – change priority of a process

C SYNOPSIS

int nice (int incr);

DESCRIPTION

nice adds the value of *incr* to the nice value of the calling process. A process's *nice value* is a non-negative number for which a more positive value results in lower CPU priority.

A maximum nice value of 39 and a minimum nice value of 0 are imposed by the system. (The default nice value is 20.) Requests for values above or below these limits result in the nice value being set to the corresponding limit.

[EPERM] *nice* will fail and not change the nice value if *incr* is negative or greater than 39 and the effective user ID of the calling process is not super-user.

SEE ALSO

exec(2), setpriority(2), schedctl(2).
nice(1), csh(1), sh(1) in the *User's Reference Manual.*

DIAGNOSTICS

Upon successful completion, *nice* returns the new nice value minus 20. Otherwise, a value of –1 is returned and *errno* is set to indicate the error.

NOTES

The *csh(1)* has a version of *nice* as a builtin command which alas, has slightly different syntax and semantics.

NAME

open – open for reading or writing

C SYNOPSIS

#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>

int open (const char *path, int oflag ...);

Type of optional third argument:
mode_t mode;

DESCRIPTION

Path points to a path name naming a file. *open* opens a file descriptor for the named file and sets the file status flags according to the value of *oflag*. For non-STREAMS [see *intro*(2)] files, *oflag* values are constructed by or-ing flags from the following list (only one of the first three flags below may be used):

O_RDONLY Open for reading only.

O_WRONLY Open for writing only.

O_RDWR Open for reading and writing.

O_NOCTTY If set, and *path* identifies a terminal device, the *open* function shall not cause the terminal device to become the controlling terminal for the process.

O_NDELAY This flag may affect subsequent reads and writes [see *read*(2) and *write*(2)].

When opening a FIFO with O_RDONLY or O_WRONLY set:

If O_NDELAY is set:

> An *open* for reading-only will return without delay. An *open* for writing-only will return an error if no process currently has the file open for reading.

If O_NDELAY is clear:

> An *open* for reading-only will block until a process opens the file for writing. An *open* for writing-only will block until a process opens the file for reading.

When opening a file associated with a communication line:

If O_NDELAY is set:

The open will return without waiting for carrier.

If O_NDELAY is clear:

The open will block until carrier is present.

O_NONBLOCK
This flag functions identically to O_NDELAY with regard to the *open* function. [See *read*(2) and *write*(2))].

O_APPEND If set, the file pointer will be set to the end of the file prior to each write.

O_SYNC When opening a regular file, this flag affects subsequent writes. If set, each *write*(2) will wait for both the file data and file status to be physically updated.

O_CREAT If the file exists, this flag has no effect. Otherwise, the owner ID of the file is set to the effective user ID of the process, the group ID of the file is set to the effective group ID of the process or to the group ID of the directory in which the file is being created. This is determined as follows:

If the underlying filesystem was mounted with the BSD file creation semantics flag [see *fstab*(4)] or the **S_ISGID** bit is set [see *chmod*(2)] on the parent directory, then the group ID of the new file is set to the group ID of the parent directory, otherwise it is set to the effective group ID of the calling process.

The low-order 12 bits of the file mode are set to the value of *mode* modified as follows [see *creat*(2)]:

All bits set in the file mode creation mask of the process are cleared [see *umask*(2)].

The "sticky bit" of the mode is cleared [see *chmod*(2)].

O_TRUNC If the file exists, its length is truncated to 0 and the mode and owner are unchanged.

O_EXCL If O_EXCL and O_CREAT are set, *open* will fail if the file exists.

When opening a STREAMS file, *oflag* may be constructed from O_NDELAY or-ed with either O_RDONLY, O_WRONLY or O_RDWR. Other flag values are not applicable to STREAMS devices and have no effect on them. The value of O_NDELAY affects the operation of STREAMS drivers and certain system calls [see *read*(2), *getmsg*(2), *putmsg*(2) and *write*(2)]. For drivers, the implementation of O_NDELAY is device-specific. Each STREAMS device driver may treat this option differently.

Certain flag values can be set following *open* as described in *fcntl*(2).

The file pointer used to mark the current position within the file is set to the beginning of the file.

The new file descriptor is set to remain open across *execve*(2) system calls [see *fcntl*(2)].

There is a system enforced limit on the number of open file descriptors per process *{OPEN_MAX}*, whose value is returned by the *getdtablesize*(2) function.

The named file is opened unless one or more of the following are true:

[EACCES] A component of the path prefix denies search permission.

[ENAMETOOLONG] The length of *path* exceeds *{PATH_MAX}*, or a pathname component is longer than *{NAME_MAX}*.

[ELOOP] Too many symbolic links were encountered in translating the pathname.

[EACCES] *oflag* permission is denied for the named file.

[EAGAIN] The file exists, mandatory file/record locking is set, and there are outstanding record locks on the file [see *chmod* (2)].

[EEXIST] O_CREAT and O_EXCL are set, and the named file exists.

[EFAULT] *Path* points outside the allocated address space of the process.

[EINTR] A signal was caught during the *open* system call.

[EIO] A hangup or error occurred during a STREAMS *open*.

[EISDIR] The named file is a directory and *oflag* is write or read/write.

[EMFILE]	The system imposed limit for open file descriptors per process *{OPEN_MAX}* has already been reached.
[ENFILE]	The system file table has exceeded *{NFILE_MAX}* concurrently open files.
[ENOENT]	O_CREAT is not set and the named file does not exist.
[ENOMEM]	The system is unable to allocate a send descriptor.
[ENOSPC]	O_CREAT and O_EXCL are set, and the file system is out of inodes.
[ENOSR]	Unable to allocate a *stream*.
[ENOTDIR]	A component of the path prefix is not a directory.
[ENXIO]	The named file is a character special or block special file, and the device associated with this special file does not exist.
[ENXIO]	O_NDELAY is set, the named file is a FIFO, O_WRONLY is set, and no process has the file open for reading.
[ENXIO]	A STREAMS module or driver open routine failed.
[EROFS]	The named file resides on a read-only file system and *oflag* is write or read/write.
[ETXTBSY]	The file is a pure procedure (shared text) file that is being executed and *oflag* is write or read/write.
[EOPNOTSUPP]	An attempt was made to open a socket (not currently supported).

SEE ALSO

chmod(2), close(2), creat(2), dup(2), fcntl(2), getdtablesize(2), intro(2), lseek(2), read(2), getmsg(2), putmsg(2), umask(2), write(2).

DIAGNOSTICS

Upon successful completion, the file descriptor is returned. Otherwise, a value of −1 is returned and *errno* is set to indicate the error.

NAME

pathconf, fpathconf – get configurable pathname variables

SYNOPSIS

#include <unistd.h>

long pathconf (char *path, int name);

long fpathconf (int fildes, int name);

DESCRIPTION

The *pathconf* and *fpathconf* functions provide a method for an application to determine the current value of a configurable limit or option (*variable*) that is associated with a file or directory.

For *pathconf*, the *path* argument points to the pathname of a file or directory. For *fpathconf*, the *fildes* argument is an open file descriptor.

The *name* argument represents the variable to be queried relative to that file or directory. The following table lists the variables to be queried (compile-time values of which appear in <*limits.h*> or <*unistd.h*>) on the left, and the *names* used to retrieve them (defined in <*unistd.h*>) on the right:

{LINK_MAX}	**_PC_LINK_MAX**
{MAX_CANON}	**_PC_MAX_CANON**
{MAX_INPUT}	**_PC_MAX_INPUT**
{NAME_MAX}	**_PC_NAME_MAX**
{PATH_MAX}	**_PC_PATH_MAX**
{_PIPE_BUF}	**_PC_PIPE_BUF**
{_POSIX_CHOWN_RESTRICTED}	**_PC_CHOWN_RESTRICTED**
{_POSIX_NO_TRUNC}	**_PC_NO_TRUNC**
{_POSIX_VDISABLE}	**_PC_VDISABLE**

pathconf and *fpathconf* will fail if one or more of the following are true:

[EINVAL] The value of the *name* argument is invalid.

pathconf:

[EACCESS] Search permission is denied for a component of the path prefix.

[EINVAL] This implementation does not support an association of the variable name with the specified file.

[ENAMETOOLONG] The length of *path* exceeds *{PATH_MAX}*, or a pathname component is longer than *{NAME_MAX}*.

[ENOENT] The named file does not exist or the *path* argument points to an empty string.

[ENOTDIR]	A component of the path prefix is not a directory.
fpathconf:	
[EBADF]	The *fildes* argument is not a valid file descriptor.
[EINVAL]	This implementation does not support an association of the variable name with the specified file.

SEE ALSO

sysconf(2), limits(4).

DIAGNOSTICS

Upon successful completion, the *pathconf* and *fpathconf* functions return the current variable value for the file or directory without changing *errno*. The value returned will never be more restrictive than the corresponding value described to the application when it was compiled with the implementation's <*limits.h*> or <*unistd.h*>.

If unsuccessful, *pathconf* and *fpathconf* return −1 and *errno* is set to the appropriate error.

NAME

pause – suspend process until signal

C SYNOPSIS

int pause (void);

DESCRIPTION

pause suspends the calling process until it receives a signal. The signal must be one that is not currently set to be ignored by the calling process.

If the signal causes termination of the calling process, *pause* will not return.

If the signal is *caught* by the calling process and control is returned from the signal-catching function [see *signal*(2)], the calling process resumes execution from the point of suspension; with a return value of −1 from *pause* and *errno* set to **EINTR.**

SEE ALSO

alarm(2), kill(2), signal(2), sigpause(2), wait(2), sigaction(2), sigpending(2), sigprocmask(2), sigsuspend(2), sigvec(3B), signal(3B), sigblock(3B), sigpause(3B), sigsetmask(3B).

NAME

pipe – create an interprocess channel

C SYNOPSIS

#include <unistd.h>

int pipe (int *fildes);

DESCRIPTION

pipe creates an I/O mechanism called a pipe and returns two file descriptors, *fildes*[0] and *fildes*[1]. *Fildes*[0] is opened for reading and *fildes*[1] is opened for writing.

Up to **PIPE_BUF** (defined in **limits.h**) are guaranteed to be written atomically. *Up to* **PIPE_MAX** (defined in **limits.h**) bytes of data are buffered by the pipe before the writing process is blocked. A read only file descriptor *fildes*[0] accesses the data written to *fildes*[1] on a first-in-first-out (FIFO) basis.

pipe will fail if:

[EMFILE] more than *{OPEN_MAX}-2* file descriptors are currently open.

[ENFILE] The system file table has exceeded *{NFILE_MAX}* concurrently open files.

SEE ALSO

read(2), write(2).
sh(1) in the *User's Reference Manual.*

DIAGNOSTICS

Upon successful completion, a value of 0 is returned. Otherwise, a value of −1 is returned and *errno* is set to indicate the error.

NAME

plock – lock process, text, or data in memory

C SYNOPSIS

#include <sys/lock.h>

int plock (int op);

DESCRIPTION

plock allows the calling process to lock its text segment (text lock), its data and stack segments (data lock), or its text, data, and stack segments (process lock) into memory. Locked segments are immune to all routine swapping. *plock* also allows these segments to be unlocked. The effective user ID of the calling process must be super-user to use this call. *Op* specifies the following:

PROCLOCK – lock text and data segments into memory (process lock)

TXTLOCK – lock text segment into memory (text lock)

DATLOCK – lock data and stack segments into memory (data lock)

UNLOCK – remove locks

plock will fail and not perform the requested operation if one or more of the following are true:

[EPERM] The effective user ID of the calling process is not super-user.

[EINVAL] *Op* is equal to **PROCLOCK** and a process lock, a text lock, or a data lock already exists on the calling process.

[EINVAL] *Op* is equal to **TXTLOCK** and a text lock or a process lock already exists on the calling process.

[EINVAL] *Op* is equal to **DATLOCK** and a data lock or a process lock already exists on the calling process.

[EINVAL] *Op* is equal to **UNLOCK** and no type of lock exists on the calling process.

[EAGAIN] There was insufficient lockable memory to lock the requested segment. This may occur even though the amount requested was less than the system-imposed maximum number of locked pages.

[ENOMEM] The caller was not super-user and the number of pages to be locked exceeded the per process limit *{PLOCK_MAX}*.

[ENOMEM] The total number of pages locked by the caller would exceed the maximum resident size for the process [see *setrlimit*(2)].

SEE ALSO

intro(2), exec(2), exit(2), getrlimit(2), mpin(2), plock(2), shmctl(2), ulimit(2).

WARNING

If a locked data segment is grown, the newly-allocated pages are not locked into memory.

DIAGNOSTICS

Upon successful completion, a value of 0 is returned to the calling process. Otherwise, a value of −1 is returned and *errno* is set to indicate the error.

BUGS

Shared library text and data segments and mapped files are not currently affected by calls to *plock*.

NAME

poll – input/output multiplexing

SYNOPSIS

```
#include <stropts.h>
#include <poll.h>

int poll(fds, nfds, timeout)
struct pollfd fds[];
unsigned long nfds;
int timeout;
```

DESCRIPTION

The IRIX version of *poll* provides users with a mechanism for multiplexing input and output over a set of any type of file descriptors, rather than the traditional limitation to only descriptors of STREAMS devices [see *select*(2)]. *Poll* identifies those descriptors on which a user can send or receive messages, or on which certain events have occurred.

Fds specifies the file descriptors to be examined and the events of interest for each file descriptor. It is a pointer to an array with one element for each open file descriptor of interest. The array's elements are *pollfd* structures which contain the following members:

```
int fd;          /* file descriptor      */
short events;    /* requested events     */
short revents;   /* returned events      */
```

where *fd* specifies an open file descriptor and *events* and *revents* are bitmasks constructed by or-ing any combination of the following event flags:

POLLIN A non-priority or file descriptor passing message (see I_RECVFD) is present on the *stream head* read queue. This flag is set even if the message is of zero length. In *revents*, this flag is mutually exclusive with POLLPRI.

POLLPRI A priority message is present on the *stream head* read queue. This flag is set even if the message is of zero length. In *revents*, this flag is mutually exclusive with POLLIN.

POLLOUT The first downstream write queue in the *stream* is not full. Priority control messages can be sent (see *putmsg*) at any time.

POLLERR An error message has arrived at the *stream head*. This flag is only valid in the *revents* bitmask; it is not used in the *events* field.

POLLHUP A hangup has occurred on the *stream*. This event and POLLOUT are mutually exclusive; a *stream* can never be writable if a hangup has occurred. However, this event and POLLIN or POLLPRI are not mutually exclusive. This flag is only valid in the *revents* bitmask; it is not used in the *events* field.

POLLNVAL The specified *fd* value does not belong to an open *stream*. This flag is only valid in the *revents* field; it is not used in the *events* field.

For each element of the array pointed to by *fds*, *poll* examines the given file descriptor for the event(s) specified in *events*. The number of file descriptors to be examined is specified by *nfds*. If *nfds* exceeds NOFILES, the system limit of open files [see *ulimit*(2)], *poll* will fail.

If the value *fd* is less than zero, *events* is ignored and *revents* is set to 0 in that entry on return from *poll*.

The results of the *poll* query are stored in the *revents* field in the *pollfd* structure. Bits are set in the *revents* bitmask to indicate which of the requested events are true. If none are true, none of the specified bits is set in *revents* when the *poll* call returns. The event flags POLLHUP, POLLERR and POLLNVAL are always set in *revents* if the conditions they indicate are true; this occurs even though these flags were not present in *events*.

If none of the defined events have occurred on any selected file descriptor, *poll* waits at least *timeout* msec for an event to occur on any of the selected file descriptors. On a computer where millisecond timing accuracy is not available, *timeout* is rounded up to the nearest legal value available on that system. If the value *timeout* is 0, *poll* returns immediately. If the value of *timeout* is -1, *poll* blocks until a requested event occurs or until the call is interrupted. *poll* is not affected by the O_NDELAY flag.

poll fails if one or more of the following are true:

[EAGAIN] Allocation of internal data structures failed but request should be attempted again.

[EFAULT] Some argument points outside the allocated address space.

[EINTR] A signal was caught during the *poll* system call.

[EINVAL] The argument *nfds* is less than zero, or *nfds* is greater than NOFILES.

SEE ALSO

intro(2), select(2), read(2), write(2), getmsg(2), putmsg(2).
streamio(7) in the *System Administrator's Reference Manual.*
STREAMS Primer.
STREAMS Programmer's Guide.

DIAGNOSTICS

Upon successful completion, a non-negative value is returned. A positive value indicates the total number of file descriptors that has been selected (i.e., file descriptors for which the *revents* field is non-zero). A value of 0 indicates that the call timed out and no file descriptors have been selected. Upon failure, a value of -1 is returned and *errno* is set to indicate the error.

NAME

prctl – operations on a process

C SYNOPSIS

#include <sys/types.h>
#include <sys/prctl.h>

int prctl (unsigned option, [void *value, [void *value2]]);

DESCRIPTION

prctl provides information about processes and the ability to control certain of their attributes. *Option* specifies one of the following actions:

PR_MAXPROCS returns the system imposed limit on the number of processes per user.

PR_MAXPPROCS returns the maximum number of processes the system is willing to run in parallel.

PR_ISBLOCKED returns 1 if the specified process is currently blocked. *value* specifies the pid. Since other processes could have subsequently unblocked the subject process, the result should be considered as a snapshot only.

PR_SETSTACKSIZE sets the maximum stack size for the current process. This affects future stack growths and forks only. The new value, suitably rounded, is returned. The value is expressed in terms of bytes. This option and the **RLIMIT_STACK** option of *setrlimit*(2) act on the same value.

PR_GETSTACKSIZE returns the current process's maximum stack size in bytes. This size is an upper limit on the size of the current process's stack.

PR_UNBLKONEXEC sets a flag so that when the calling process subsequently calls *exec*(2), the process whose pid is specified by *value* is unblocked. This can be used in conjunction with the **PR_BLOCK** option of *sproc*(2) to provide race-free process creation.

PR_SETEXITSIG controls whether all members of a share group will be signaled if any one of them terminates. If *value* is 0, then normal **IRIX** process termination rules apply, namely that the parent is sent a **SIGCLD** upon death of child, but no indication of death of parent is given. If *value* is a valid signal number [see *signal*(2)] then if any member of a share group terminates, that signal is sent to ALL surviving members of the share group.

PR_RESIDENT makes the process immune to process swapout.

PR_ATTACHADDR attaches the virtual segment containing the address *value2* in the process whose pid is *value* to the calling process. Both processes must be members of the same share group. The address of where the virtual segment was attached is returned. This address has the same logical offset into the virtual space as the passed in address.

prctl will fail if one or more of the following are true:

[EINVAL] *option* is not valid.

[ESRCH] The *value* passed with the **PR_ISBLOCKED** or **PR_UNBLKONEXEC** option doesn't match the *pid* of any process.

[EINVAL] The value given for the new maximum stack size is negative or exceeds the maximum process size allowed.

[EINVAL] The value given for the **PR_SETEXITSIG** option is not a valid signal number.

[EINVAL] The calling process already has specified a process (or the specified process is the caller itself) to be unblocked on exec via the **PR_UNBLKONEXEC** option.

[EPERM] The caller does not have permission to unblock the process specified by the *value* passed for the **PR_UNBLKONEXEC** option.

SEE ALSO

blockproc(2), signal(2), setrlimit(2), sproc(2).

DIAGNOSTICS

Upon successful completion, *prctl* returns the requested information. Otherwise, a value of −1 is returned to the calling process, and *errno* is set to indicate the error.

NAME

profil – execution time profile

C SYNOPSIS

profil(buff, bufsiz, offset, scale)
short *buff;
int bufsiz, offset, scale;

DESCRIPTION

Buff points to an area of core whose length (in bytes) is given by *bufsiz*. After this call, the user's program counter (pc) is examined each clock tick (10 milliseconds); *offset* is subtracted from it, and the result multiplied by *scale*. If the resulting number corresponds to a word inside *buff*, that word is incremented.

The scale is interpreted as an unsigned, fixed-point fraction with 16 bits of fraction: 0x10000 gives a 1-1 mapping of pc's to words in *buff;* 0x8000 maps each pair of instruction words together.

Since each bucket is only 16 bits, it is concievable for it to overflow. No indication that this has occurred is given.

Profiling is turned off by giving a *scale* of 0 or 1. It is rendered ineffective by giving a *bufsiz* of 0. Profiling is turned off when an *execve* is executed, but remains on in child and parent both after a *fork* or *sproc*. Profiling is turned off if an update in *buff* would cause a memory fault.

SEE ALSO

fork(2), sproc(2), monitor(3X).

DIAGNOSTICS

A 0, indicating success, is always returned.

NAME

ptrace – process trace

C SYNOPSIS

#include <signal.h>
#include <ptrace.h>

ptrace (request, pid, addr, data)
int request, pid, *addr, data;

DESCRIPTION

Ptrace provides a way for a process to control the execution of another process and to examine and change that process's core image. *Ptrace* is used primarily to implement breakpoint debugging.

There are four arguments whose interpretation depends on a *request* argument. Generally, *pid* is the process ID of the traced process. A process being traced behaves normally until it encounters some signal, whether internally generated (for example, ''illegal instruction'')) or externally generated (for example, ''interrupt''). See *signal*(2) for the list.

When the traced process encounters a signal, it enters a stopped state. The process tracing it is notified by *wait*(2). If the the traced process stops with a SIGTRAP, the process might have stopped for many reasons. Two status words, which are addressable as registers, in the traced process's uarea qualify SIGTRAPs: TRAPCAUSE, which contains the cause of the trap, and TRAPINFO, which contains extra information about the trap.

When the traced process is in the stopped state, its core image can be examined and modified using *ptrace*. Another *ptrace* request can cause the traced process either to terminate or to continue, possibly ignoring the signal.

The value of the *request* argument determines the precise action of the call:

0 This request is the only one that can be used by a child process. Request 0 can declare that the child process is to be traced by its parent. All other arguments are ignored. Peculiar results happen when the parent does not expect to trace the child.

1,2 The word in the traced process's address space at *addr* is returned. If I and D space are separated (for example, historically on a PDP-11), Request 1 specifies I space and Request 2 specifies D space. *Addr* must be 4-byte aligned. The traced process must be stopped. The input *data* is ignored.

3 The word of the system's per-process data area that corresponds to *addr* is returned. *Addr* is a constant defined in ptrace.h. This space contains the registers and other information about the process. The constants correspond to fields in the system's *user* structure.

4,5 The specified *data* is written at the word in the process's address space corresponds to *addr*. *Addr* must be 4-byte aligned. Upon successful completion, the value written into the address space of the child is returned to the parent. If I and D space are separated, Request 4 specifies I space and Request 5 specifies D space. Attempts to write in pure procedure fail when another process is executing the same file.

6 The process's system data is written as it is read with Request 3. Only a few locations can be written this way: the general registers, the floating point status and registers, and certain bits of the processor status word. The old value at the address is returned.

7 The *data* argument is taken as a signal number and the traced process's execution continues at location *addr* as if it had incurred that signal. The signal number can be 0, indicating the signal that caused the stop should be ignored, or the signal can be the value fetched from the process's image, indicating what signal caused the stop. If *addr* is (int *)1, execution continues from where it stopped.

8 The traced process terminates. The *addr* argument is ignored and the *data* argument is the signal specified in Request 7.

9 Execution continues as in Request 7; however, as soon as possible after execution of at least one instruction, execution stops again. The signal number from the stop is SIGTRAP. TRAPCAUSE contains CAUSESINGLE. This part of *ptrace* is used to implement breakpoints. The *addr* and *data* arguments are defined in Request 7.

Requests 20-29 have not been fully defined or implemented.

20 This request is the same as Request 0, except it is executed by the tracing process and the pid field is non-zero. That pid's process pid stops execution. On a signal, it becomes a traced process that returns control to the tracing process rather than to the parent. The tracing process must have the same user-id (uid) as the traced process.

21,22
These requests return MAXREG general registers or MAXFREG floating registers, respectively. Their values are copied to the locations starting at the address in the tracing process specified by the *addr* argument.

24,25
These requests are the same as Requests 20 and 21, except that they write the registers instead of reading them.

26 This request specifies a watchpoint in the data or stack segment of the traced process. If any byte address (starting at the *addr* argument and continuing for the number of bytes specified by the *data* argument) is accessed in an instruction, the traced process stops execution with a SIGTRAP. TRAPCAUSE contains CAUSEWATCH; TRAPINFO contains the address causing the trap. *Ptrace* returns a wid (watchpoint identifier). MAXWIDS specifies the maximum number of watchpoints per process.

27 This request's data argument specifies a wid to delete.

28 This request turns off tracing for the traced process that has the specified pid.

29 This request returns an open file descriptor for the file attached to pid. This is useful for accessing the symbol table of an execed process.

These calls (except for Requests 0 and 20) can be used only when the subject process has terminated. The *wait* call determines when a process terminates. Then, the "termination" status returned by *wait* has the value 0177 to show stoppage rather than termination. If multiple processes are being traced, **wait** can be called multiple times and returns the status for the next stopped child, terminated child, or traced process.

To prevent fraud, *ptrace* inhibits the set-user-id and set-group-id facilities on later *execve*(2) calls. If a traced process calls *execve*, the process terminates before executing the first instruction of the new image showing signal SIGTRAP. TRAPCAUSE contains CAUSEEXEC; TRAPINFO does not contain anything interesting. If a traced process execs again, the same thing happens.

If a traced process forks, both parent and child are traced, and the breakpoints from the parent are copied into the child. At the time of the fork, the child stops with a SIGTRAP. The tracing process can end the trace, if desired. TRAPCAUSE contains CAUSEFORK; TRAPINFO contains the its parent's pid.

RETURN VALUE

If the call succeeds, a 0 value is returned. If the call fails, then a −1 is returned and the global variable *errno* is set to indicate the error.

ERRORS

[EINVAL]	The request code is invalid.
[EINVAL]	The specified process does not exist.
[EINVAL]	The given signal number is invalid.
[EFAULT]	The specified address is out of bounds.
[EPERM]	The specified process cannot be traced.

SEE ALSO

wait(2), sigvec(2).

BUGS

There is file system called /debug where each "file" is actually an active process. The process' file name is /debug/processid where processid is the process number. *open*(2), *read*(2), etc can be used to acess the (running) process. Use *fcntl*(2) to control the process. See <*sys/fs/dbfcntl.h*> for a list of the control functions available. The /debug facility solves the problems with *ptrace* mentioned below.

Ptrace is unique and arcane; it should be replaced with a special file that can be opened, read, and written. The control functions could be implemented with *ioctl*(2) calls on this file. This would be easier to understand and have much higher performance.

The Request 0 call should specify signals that are to be treated normally and should not cause a termination. Then, programs with simulated floating point (which use "illegal instruction" signals at a high rate) could be efficiently debugged.

The error indication −1 is a legitimate function value *errno*. See *intro*(2) to disambiguate.

It should be possible to stop a process on occurrence of a system call. In this way, a completely controlled environment could be provided.

NAME

putmsg – send a message on a stream

SYNOPSIS

#include <stropts.h>

int putmsg (fd, ctlptr, dataptr, flags)
int fd;
struct strbuf *ctlptr;
struct strbuf *dataptr;
int flags;

DESCRIPTION

putmsg creates a message [see *intro*(2)] from user specified buffer(s) and sends the message to a STREAMS file. The message may contain either a data part, a control part or both. The data and control parts to be sent are distinguished by placement in separate buffers, as described below. The semantics of each part is defined by the STREAMS module that receives the message.

fd specifies a file descriptor referencing an open *stream. ctlptr* and *dataptr* each point to a *strbuf* structure which contains the following members:

```
int maxlen;     /* not used      */
int len;        /* length of data */
char *buf;      /* ptr to buffer  */
```

ctlptr points to the structure describing the control part, if any, to be included in the message. The *buf* field in the *strbuf* structure points to the buffer where the control information resides, and the *len* field indicates the number of bytes to be sent. The *maxlen* field is not used in *putmsg* [see *getmsg*(2)]. In a similar manner, *dataptr* specifies the data, if any, to be included in the message. *flags* may be set to the values 0 or RS_HIPRI and is used as described below.

To send the data part of a message, *dataptr* must be non-NULL and the *len* field of *dataptr* must have a value of 0 or greater. To send the control part of a message, the corresponding values must be set for *ctlptr*. No data (control) part will be sent if either *dataptr* (*ctlptr*) is NULL or the *len* field of *dataptr* (*ctlptr*) is set to -1.

If a control part is specified, and *flags* is set to RS_HIPRI, a *priority* message is sent. If *flags* is set to 0, a non-priority message is sent. If no control part is specified, and *flags* is set to RS_HIPRI, *putmsg* fails and sets *errno* to EINVAL. If no control part and no data part are specified, and *flags* is set to 0, no message is sent, and 0 is returned.

For non-priority messages, *putmsg* will block if the *stream* write queue is full due to internal flow control conditions. For priority messages, *putmsg* does not block on this condition. For non-priority messages, *putmsg* does not block when the write queue is full and O_NDELAY is set. Instead, it fails and sets *errno* to EAGAIN.

putmsg also blocks, unless prevented by lack of internal resources, waiting for the availability of message blocks in the *stream*, regardless of priority or whether O_NDELAY has been specified. No partial message is sent.

putmsg fails if one or more of the following are true:

[EAGAIN] A non-priority message was specified, the O_NDELAY flag is set and the *stream* write queue is full due to internal flow control conditions.

[EAGAIN] Buffers could not be allocated for the message that was to be created.

[EBADF] *fd* is not a valid file descriptor open for writing.

[EFAULT] *ctlptr* or *dataptr* points outside the allocated address space.

[EINTR] A signal was caught during the *putmsg* system call.

[EINVAL] An undefined value was specified in *flags*, or *flags* is set to RS_HIPRI and no control part was supplied.

[EINVAL] The *stream* referenced by *fd* is linked below a multiplexor.

[ENOSTR] A *stream* is not associated with *fd*.

[ENXIO] A hangup condition was generated downstream for the specified *stream*.

[ERANGE] The size of the data part of the message does not fall within the range specified by the maximum and minimum packet sizes of the topmost *stream* module. This value is also returned if the control part of the message is larger than the maximum configured size of the control part of a message, or if the data part of a message is larger than the maximum configured size of the data part of a message.

A *putmsg* also fails if a STREAMS error message had been processed by the *stream* head before the call to *putmsg*. The error returned is the value contained in the STREAMS error message.

SEE ALSO

intro(2), read(2), getmsg(2), poll(2), write(2).
STREAMS Primer.
STREAMS Programmer's Guide.

DIAGNOSTICS

Upon successful completion, a value of 0 is returned. Otherwise, a value of -1 is returned and *errno* is set to indicate the error.

NAME

read – read from file

C SYNOPSIS

#include <unistd.h>

int read (int fildes, void *buf, unsigned nbyte);

DESCRIPTION

Fildes is a file descriptor obtained from a *creat*(2), *open*(2), *dup*(2), *fcntl*(2), *socket*(2), *socketpair*(2), or *pipe*(2) system call.

read attempts to read *nbyte* bytes from the file associated with *fildes* into the buffer pointed to by *buf*.

On devices capable of seeking, the *read* starts at a position in the file given by the file pointer associated with *fildes*. Upon return from *read*, the file pointer is incremented by the number of bytes actually read.

Devices that are incapable of seeking always read from the current position. The value of a file pointer associated with such a file is undefined.

Upon successful completion, *read* returns the number of bytes actually read and placed in the buffer; this number may be less than *nbyte* if the file is associated with a communication line [see *ioctl*(2) and *termio*(7)], or a socket [see *socket*(2)], or if the number of bytes left in the file is less than *nbyte* bytes. A value of 0 is returned when an end-of-file has been reached.

A *read* from a STREAMS [see *intro*(2)] file can operate in three different modes: "byte-stream" mode, "message-nondiscard" mode, and "message-discard" mode. The default is byte-stream mode. This can be changed using the I_SRDOPT *ioctl* request [see *streamio*(7)], and can be tested with the I_GRDOPT *ioctl*. In byte-stream mode, *read* will retrieve data from the *stream* until it has retrieved *nbyte* bytes, or until there is no more data to be retrieved. Byte-stream mode ignores message boundaries.

In STREAMS message-nondiscard mode, *read* retrieves data until it has read *nbyte* bytes, or until it reaches a message boundary. If the *read* does not retrieve all the data in a message, the remaining data are replaced on the *stream*, and can be retrieved by the next *read* or *getmsg*(2) call. Message-discard mode also retrieves data until it has retrieved *nbyte* bytes, or it reaches a message boundary. However, unread data remaining in a message after the *read* returns are discarded, and are not available for a subsequent *read* or *getmsg*.

When attempting to read from a regular file with mandatory file/record locking set [see *chmod*(2)], and there is a blocking (i.e. owned by another process) write lock on the segment of the file to be read:

If O_NDELAY or O_NONBLOCK is set, the read will return a -1 and set errno to EAGAIN.

If O_NDELAY and O_NONBLOCK are clear, the read will sleep until the blocking record lock is removed.

When attempting to read from an empty pipe (or FIFO):

If O_NDELAY is set, the read will return a 0.

If O_NONBLOCK is set, the read will return a -1 and set errno to EAGAIN.

If O_NDELAY and O_NONBLOCK are clear, the read will block until data is written to the file or the file is no longer open for writing.

When attempting to read a file associated with a tty that has no data currently available:

If O_NDELAY is set, the read will return a 0.

If O_NONBLOCK is set, the read will return a -1 and set errno to EAGAIN.

If O_NDELAY and O_NONBLOCK are clear, the read will block until data becomes available.

When attempting to read a file associated with a *stream* that has no data currently available:

If O_NDELAY or O_NONBLOCK is set, the read will return a -1 and set errno to EAGAIN.

If O_NDELAY and O_NONBLOCK are clear, the read will block until data becomes available.

Due to the different semantics of O_NDELAY and O_NONBLOCK in two of the above 4 cases, these flags *must* not be used simultaneously.

When reading from a STREAMS file, handling of zero-byte messages is determined by the current read mode setting. In byte-stream mode, *read* accepts data until it has read *nbyte* bytes, or until there is no more data to read, or until a zero-byte message block is encountered. *read* then returns the number of bytes read, and places the zero-byte message back on the *stream* to be retrieved by the next *read* or *getmsg*. In the two other modes, a zero-byte message returns a value of 0 and the message is removed from the *stream.* When a zero-byte message is read as the first message on a *stream*, a value of 0 is returned regardless of the read mode.

A *read* from a STREAMS file can only process data messages. It cannot process any type of protocol message and will fail if a protocol message is encountered at the *stream head.*

read will fail if one or more of the following are true:

[EAGAIN] Mandatory file/record locking was set, O_NDELAY was set, and there was a blocking record lock.

[ENOMEM] Insufficient amount of system virtual memory is available with which to map the user pages when reading via raw IO.

[EAGAIN] No message waiting to be read on a *stream* and O_NDELAY flag set.

[EBADF] *Fildes* is not a valid file descriptor open for reading.

[EBADMSG] Message waiting to be read on a *stream* is not a data message.

[EDEADLK] The read was going to go to sleep and cause a deadlock situation to occur.

[EFAULT] *Buf* points outside the allocated address space.

[EINTR] A signal was caught during the *read* system call.

[EINVAL] Attempted to read from a *stream* linked to a multiplexor.

[ENOLCK] The system record lock table was full, so the read could not go to sleep until the blocking record lock was removed.

A *read* from a STREAMS file will also fail if an error message is received at the *stream head.* In this case, *errno* is set to the value returned in the error message. If a hangup occurs on the *stream* being read, *read* will continue to operate normally until the *stream head* read queue is empty. Thereafter, it will return 0.

SEE ALSO

creat(2), dup(2), fcntl(2), ioctl(2), intro(2), open(2), pipe(2), getmsg(2), socket(2).
streamio(7), termio(7) in the *System Administrator's Reference Manual.*

DIAGNOSTICS

Upon successful completion a non-negative integer is returned indicating the number of bytes actually read. Otherwise, a −1 is returned and *errno* is set to indicate the error.

NAME

readlink – read value of a symbolic link

C SYNOPSIS

cc = readlink(path, buf, bufsiz)
int cc;
char *path, *buf;
int bufsiz;

DESCRIPTION

Readlink places the contents of the symbolic link *path* in the buffer *buf* which has size *bufsiz*. The contents of the link are not null terminated when returned.

RETURN VALUE

The call returns the count of characters placed in the buffer if it succeeds, or a –1 if an error occurs, placing the error code in the global variable *errno*.

ERRORS

Readlink will fail and the file mode will be unchanged if:

[ENOTDIR]	A component of the path prefix is not a directory.
[ENOENT]	The named file does not exist.
[ENXIO]	The named file is not a symbolic link.
[EACCES]	Search permission is denied on a component of the path prefix.
[EFAULT]	*Buf* extends outside the process's allocated address space.
[ELOOP]	Too many symbolic links were encountered in translating the pathname.

SEE ALSO

stat(2), symlink(2)

NAME

recv, recvfrom, recvmsg – receive a message from a socket

SYNOPSIS

```
#include <sys/types.h>
#include <sys/socket.h>

cc = recv(s, buf, len, flags)
int cc, s;
char *buf;
int len, flags;

cc = recvfrom(s, buf, len, flags, from, fromlen)
int cc, s;
char *buf;
int len, flags;
struct sockaddr *from;
int *fromlen;

cc = recvmsg(s, msg, flags)
int cc, s;
struct msghdr msg[];
int flags;
```

DESCRIPTION

Recv, *recvfrom*, and *recvmsg* are used to receive messages from a socket.

The *recv* call is normally used only on a *connected* socket (see *connect*(2)), while *recvfrom* and *recvmsg* may be used to receive data on a socket whether it is in a connected state or not.

If *from* is non-zero, the source address of the message is filled in. *Fromlen* is a value-result parameter, initialized to the size of the buffer associated with *from*, and modified on return to indicate the actual size of the address stored there. The length of the message is returned in *cc*. If a message is too long to fit in the supplied buffer, excess bytes may be discarded depending on the type of socket the message is received from (see *socket*(2)).

If no messages are available at the socket, the receive call waits for a message to arrive, unless the socket is nonblocking (see *ioctl*(2)) in which case a *cc* of −1 is returned with the external variable errno set to EWOULDBLOCK.

The *select*(2) call may be used to determine when more data arrives.

The *flags* argument to a recv call is formed by *or*'ing one or more of the values,

```
#define     MSG_OOB     0x1     /* process out-of-band data */
#define     MSG_PEEK    0x2     /* peek at incoming message */
```

The *recvmsg* call uses a *msghdr* structure to minimize the number of directly supplied parameters. This structure has the following form, as defined in <*sys/socket.h*>:

```
struct msghdr {
        caddr_t msg_name;       /* optional address */
        int     msg_namelen;    /* size of address */
        struct  iovec *msg_iov; /* scatter/gather array */
        int     msg_iovlen;     /* # elements in msg_iov */
        caddr_t msg_accrights;  /* access rights sent/received */
        int     msg_accrightslen;
};
```

Here *msg_name* and *msg_namelen* specify the destination address if the socket is unconnected; *msg_name* may be given as a null pointer if no names are desired or required. The *msg_iov* and *msg_iovlen* describe the scatter gather locations, as described in *read*(2). A buffer to receive any access rights sent along with the message is specified in *msg_accrights*, which has length *msg_accrightslen*. Access rights are currently limited to file descriptors, which each occupy the size of an **int**.

RETURN VALUE

These calls return the number of bytes received, or −1 if an error occurred.

ERRORS

The calls fail if:

[EBADF]	The argument *s* is an invalid descriptor.
[ENOTSOCK]	The argument *s* is not a socket.
[EWOULDBLOCK]	The socket is marked non-blocking and the receive operation would block.
[EINTR]	The receive was interrupted by delivery of a signal before any data was available for the receive.
[EFAULT]	The data was specified to be received into a non-existent or protected part of the process address space.

SEE ALSO

fcntl(2), getsockopt(2), read(2), select(2), send(2), socket(2)

NAME

rename – change the name of a file

SYNOPSIS

#include <unistd.h>
#include <stdio.h>

int rename(const char *from, const char *to);

DESCRIPTION

rename causes the link named *from* to be renamed as *to*. If *to* exists, then it is first removed. Both *from* and *to* must be of the same type (that is, both directories or both non-directories), and must reside on the same file system.

rename guarantees that an instance of *to* will always exist, even if the system should crash in the middle of the operation.

If the final component of *from* is a symbolic link, the symbolic link is renamed, not the file or directory to which it points.

CAVEAT

The system can deadlock if a loop in the file system graph is present. This loop takes the form of an entry in directory ''a'', say ''a/foo'', being a hard link to directory ''b'', and an entry in directory ''b'', say ''b/bar'', being a hard link to directory ''a''. When such a loop exists and two separate processes attempt to perform ''rename a/foo b/bar'' and ''rename b/bar a/foo'', respectively, the system may deadlock attempting to lock both directories for modification. Hard links to directories should be replaced by symbolic links by the system administrator.

rename will fail and neither of the argument files will be affected if any of the following are true:

[ENAMETOOLONG] The length of either *from* or *to* exceeds *{PATH_MAX}*, or a pathname component is longer than *{NAME_MAX}*.

[ENOENT] The file named by *from* does not exist.

[ENOENT] A component of the path prefix of either *from* or *to* does not exist.

[EACCES] A component of either path prefix denies search permission.

[EACCES] The requested link requires writing in a directory with a mode that denies write permission. If a rename request relocates a directory in the hierarchy, write permission in the directory to be moved is needed, since its entry for the parent directory (..) must be updated.

[EACCES] The parent directory of *to* has the sticky bit set and a file exists in that directory that would be unlinked as the result of the rename and
the parent directory is not owned by the user and
the file that would be unlinked is not owned by the user and
the file that would be unlinked is not writable by the user and
the user is not superuser.

[EACCES] The parent directory of *from* has the sticky bit set and
the parent directory is not owned by the user and
from is not owned by the user and
from is not writable by the
the user is not superuser.

[EPERM] The directory containing *from* is marked sticky, and neither the containing directory nor *from* are owned by the effective user ID.

[EPERM] The *to* file exists, the directory containing *to* is marked sticky, and neither the containing directory nor *to* are owned by the effective user ID.

[ELOOP] Too many symbolic links were encountered in translating either pathname.

[ENOTDIR] A component of either path prefix is not a directory.

[ENOTDIR] *from* is a directory, but *to* is not a directory.

[EISDIR] *to* is a directory, but *from* is not a directory.

[EXDEV] The link named by *to* and the file named by *from* are on different logical devices (file systems). Note that this error code will not be returned if the implementation permits cross-device links.

[ENOSPC] The directory in which the entry for the new name is being placed cannot be extended because there is no space left on the file system containing the directory.

[EIO] An I/O error occurred while making or updating a directory entry.

[EROFS] The requested link requires writing in a directory on a read-only file system.

[EFAULT] *path* points outside the process's allocated address space.

[EINVAL] *from* is a parent directory of *to*, or an attempt is made to rename "." or "..".

[ENOTEMPTY] *to* is a directory and is not empty.

SEE ALSO

chmod(2), link(2), open(2), unlink(2).

DIAGNOSTICS

A 0 value is returned if the operation succeeds, otherwise *rename* returns −1 and the global variable *errno* indicates the reason for the failure.

NAME

rmdir – remove a directory

C SYNOPSIS

#include <unistd.h>

int rmdir (const char *path);

DESCRIPTION

rmdir removes the directory named by the path name pointed to by *path*. The directory must not have any entries other than "." and "..".

The named directory is removed unless one or more of the following are true:

[EINVAL]	The current directory may not be removed.
[EINVAL]	The "." entry of a directory may not be removed.
[EEXIST]	The directory contains entries other than those for "." and "..".
[ENOTDIR]	A component of the path prefix is not a directory.
[ENOENT]	The named directory does not exist.
[EACCES]	Search permission is denied for a component of the path prefix.
[EACCES]	Write permission is denied on the directory containing the directory to be removed.
[EACCES]	The parent directory of the directory to be removed has the sticky bit set and the parent directory is not owned by the user and the directory to be removed is not owned by the user and the directory to be removed is not writable by the user and the user is not superuser.
[ENAMETOOLONG]	The length of *path* exceeds *{PATH_MAX}*, or a path-name component is longer than *{NAME_MAX}*.
[EBUSY]	The directory to be removed is the mount point for a mounted file system.
[EROFS]	The directory entry to be removed is part of a read-only file system.

[EFAULT] *Path* points outside the process's allocated address space.

[EIO] An I/O error occurred while accessing the file system.

DIAGNOSTICS

Upon successful completion, a value of 0 is returned. Otherwise, a value of −1 is returned and *errno* is set to indicate the error.

SEE ALSO

mkdir(2).
rmdir(1), rm(1), and mkdir(1) in the *User's Reference Manual.*

NAME

schedctl – scheduler control call

SYNOPSIS

#include <sys/types.h>
#include <sys/prctl.h>
#include <sys/schedctl.h>

int schedctl (int cmd, int arg1, int arg2);

DESCRIPTION

This system call is used to alter scheduling parameters of either individual processes or of the system as a whole. The following commands are supported:

NDPRI This command sets or removes a non-degrading priority from a process. *Arg1* is the process ID of the process to be altered; if 0, it indicates the current process. *Arg2* is the non-degrading priority. This priority may range from **NDPHIMAX** to **NDPLOMIN** (defined in the file **/usr/include/sys/schedctl.h**). If *arg2* is zero, then the non-degrading priority associated with the indicated process is removed.

A non-degrading priority is not affected by normal priority aging schemes within the kernel. Non-degrading priorities fall into three classes, with a range of priorities for each class. If the priority is between **NDPHIMAX** and **NDPHIMIN** (inclusive) then the process with such a priority is of higher priority than all other processes subject to normal scheduling. Because of this, and because the priority does not change over time, such a process may **deadlock** the system if it enters an infinite loop. If the priority is between **NDPNORMMAX** and **NDPNORMMIN** (inclusive) then the process with such a priority competes with processes subject to normal priority aging scheduling. If the priority is between **NDPLOMAX** and **NDPLOMIN** (inclusive) then the process with such a priority is of lower priority than all other processes subject to normal scheduling. These low priorities are useful for batch jobs that shouldn't take precedence over any interactive job. If multiple processes are set at the same non-degrading priority, the kernel performs round-robin scheduling between the processes at the current time-slice interval.

The process must have super-user permissions to set a non-degrading priority or to change the priority of another process. A non super-user process may only remove its own non-degrading priority. Processes created with the *fork*(2) system call inherit the non-degrading priority of the parent. The previous non-degrading

priority is returned.

RENICE This command allows a process to change its own or another process's *nice* value. *Arg1* is the process ID of the process to be modified; if zero, it indicates the current process. *Arg2* is the new process nice value to use. This is different than the value given to the *nice*(2) system call. *nice* takes a relative value, while this command changes the absolute nice value of the process which ranges from 0 to 40. The default absolute nice value for a process is 20.

The process must have super-user permissions to use this command. The previous (absolute) nice value is returned.

SLICE This command allows a process to change its own or another process's time slice. A time slice is the period of time that a process is allowed to run before being eligible for preemption by another process. *Arg1* is the process ID of the process to be altered; if zero, it indicates the current process. *Arg2* is the new time slice value to use, expressed in terms of *clock ticks*. The system software clock fires **HZ** times per second; hence the duration of a clock tick in milliseconds is equal to **1000/HZ** (see **/usr/include/sys/param.h**). *Arg2* is constrained to be greater than 0, and less than 10 seconds.

The process must have super-user permissions to use this command. The previous time slice value is returned.

The next three commands provide control over the scheduling of groups of parallel processes on multi-cpu systems. The processes must be members of the same *share group* (see *sproc*(2) for more information about share groups). Note that the **SCHEDMODE** and **SETMASTER** commands can only be used after a share group has been created.

SCHEDMODE

This command allows a member of a share group to set a *scheduling mode* for the entire share group. *Arg1* specifies the scheduling mode. These are **SGS_FREE**, which specifies that each member of the share group is to be scheduled independently, **SGS_SINGLE**, which specifies that only the *master* is to run (see **SETMASTER** for setting the master thread), and **SGS_GANG**, which specifies that all members of the share group are to be scheduled as a unit, if possible. The default scheduling mode when a share group is created is **SGS_FREE**.

The previous scheduling mode is returned.

SETMASTER

This command sets the *master process* of the share group. *Arg1* specifies the *pid* of the new master process.

By default, the creator of the share group is the master process. The master process differs from other members of the share group only in the case of the **SGS_SINGLE** scheduling mode. In that case, only the master process will run. This operation can only be performed by the master of the share group. On success, 0 is returned.

SETHINTS

This command sets up a communication path between the process and the kernel, which allows the process to communicate scheduling modes to the kernel without the overhead of a system call. The return value is a pointer to a *prda_sys* structure, defined in **/usr/include/sys/prctl.h**. Since the return value for schedctl is defined as an integer, it is necessary to cast this value to a *struct prda* * before using it.

After a **SETHINTS** command, the process may write scheduling modes to the *t_hint* field of the *prda_sys* structure. These scheduling modes will be observed by the kernel at process dispatch time. The scheduling modes are the same as those defined for the **SCHEDMODE** command.

schedctl will fail if any of the following are true:

[EINVAL] An invalid command or new value was passed to the system.

[EINVAL] The command was **SCHEDMODE**, and either the process was not a member of a share group, or *arg1* did not specify a valid scheduling mode.

[EINVAL] The command was **SETHASTER**, and either the process was not a member of a share group, the process was not the current master of the share group, or *arg1* specified a process that was not a member of the share group.

[EPERM] An attempt was made to perform privileged operations without appropriate permissions.

[ESRCH] The named process was not found.

SEE ALSO

npri(1), nice(2), prctl(2), sproc(2).

DIAGNOSTICS

schedctl returns -1 if an error occurred. Otherwise, the return is dependent on *cmd*.

NAME

select – synchronous I/O multiplexing

SYNOPSIS

#include <sys/types.h>
#include <sys/time.h>

nfound = select(nfds, readfds, writefds, exceptfds, timeout)
int nfound, nfds;
fd_set *readfds, *writefds, *exceptfds;
struct timeval *timeout;

FD_SET(fd, &fdset)
FD_CLR(fd, &fdset)
FD_ISSET(fd, &fdset)
FD_ZERO(&fdset)
int fd;
fd_set fdset;

DESCRIPTION

Select examines the I/O descriptor sets whose addresses are passed in *readfds*, *writefds*, and *exceptfds* to see if some of their descriptors are ready for reading, are ready for writing, or have an exceptional condition pending, respectively. The first *nfds* descriptors are checked in each set; i.e. the descriptors from 0 through *nfds* – 1 in the descriptor sets are examined. On return, *select* replaces the given descriptor sets with subsets consisting of those descriptors that are ready for the requested operation. The total number of ready descriptors in all the sets is returned in *nfound*.

The descriptor sets are stored as bit fields in arrays of integers. The following macros are provided for manipulating such descriptor sets: *FD_ZERO(&fdset)* initializes a descriptor set *fdset* to the null set. *FD_SET(fd, &fdset)* includes a particular descriptor *fd* in *fdset*. *FD_CLR(fd, &fdset)* removes *fd* from *fdset*. *FD_ISSET(fd, &fdset)* is nonzero if *fd* is a member of *fdset*, zero otherwise. The behavior of these macros is undefined if a descriptor value is less than zero or greater than or equal to *FD_SETSIZE*, which is normally at least equal to the maximum number of descriptors supported by the system.

If *timeout* is a non-zero pointer, it specifies a maximum interval to wait for the selection to complete. If *timeout* is a zero pointer, the select blocks indefinitely. To effect a poll, the *timeout* argument should be non-zero, pointing to a zero-valued timeval structure.

Any of *readfds*, *writefds*, and *exceptfds* may be given as zero pointers if no descriptors are of interest.

RETURN VALUE

Select returns the number of ready descriptors that are contained in the descriptor sets, or −1 if an error occurred. If the time limit expires then *select* returns 0. If *select* returns with an error, including one due to an interrupted call, the descriptor sets will be unmodified.

ERRORS

An error return from *select* indicates:

[EBADF]	One of the descriptor sets specified an invalid descriptor.
[EINTR]	A signal was delivered before the time limit expired and before any of the selected events occurred.
[EINVAL]	The specified time limit is invalid. One of its components is negative or too large.

SEE ALSO

accept(2), connect(2), read(2), write(2), recv(2), send(2).

BUGS

The dimension of a sufficiently large bit field for select remains a problem. The default size FD_SETSIZE (currently 256) is somewhat larger than the current kernel limit to the number of open files. However, in order to accommodate programs which might potentially use a larger number of open files with select, it is possible to increase this size within a program by providing a larger definition of FD_SETSIZE before the inclusion of <sys/types.h>.

Select should probably return the time remaining from the original timeout, if any, by modifying the time value in place. This may be implemented in future versions of the system. Thus, it is unwise to assume that the timeout value will be unmodified by the *select* call.

NAME

semctl – semaphore control operations

SYNOPSIS

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/sem.h>

int semctl (int semid, int semnum, int cmd, ...);
union semun {
    int val;
    struct semid_ds *buf;
    ushort *array;
} arg;
```

DESCRIPTION

semctl provides a variety of semaphore control operations as specified by *cmd*.

The following *cmd*s are executed with respect to the semaphore specified by *semid* and *semnum:*

GETVAL Return the value of semval [see *intro*(2)]. {READ}

SETVAL Set the value of semval to *arg.val*. {ALTER} When this cmd is successfully executed, the semadj value corresponding to the specified semaphore in all processes is cleared.

GETPID Return the value of sempid. {READ}

GETNCNT Return the value of semncnt. {READ}

GETZCNT Return the value of semzcnt. {READ}

The following *cmd*s return and set, respectively, every semval in the set of semaphores.

GETALL Place semvals into array pointed to by *arg.array*. {READ}

SETALL Set semvals according to the array pointed to by *arg.array*. {ALTER} When this cmd is successfully executed the semadj values corresponding to each specified semaphore in all processes are cleared.

The following *cmd*s are also available:

IPC_STAT Place the current value of each member of the data structure associated with *semid* into the structure pointed to by *arg.buf*. The contents of this structure are defined in *intro*(2). {READ}

IPC_SET Set the value of the following members of the data structure associated with *semid* to the corresponding value found in the structure pointed to by *arg.buf*:
sem_perm.uid
sem_perm.gid
sem_perm.mode /* only low 9 bits */

This cmd can only be executed by a process that has an effective user ID equal to either that of super-user, or to the value of **sem_perm.cuid** or **sem_perm.uid** in the data structure associated with *semid*.

IPC_RMID Remove the semaphore identifier specified by *semid* from the system and destroy the set of semaphores and data structure associated with it. This cmd can only be executed by a process that has an effective user ID equal to either that of super-user, or to the value of **sem_perm.cuid** or **sem_perm.uid** in the data structure associated with *semid*.

semctl fails if one or more of the following are true:

[EINVAL] *Semid* is not a valid semaphore identifier.

[EINVAL] *Semnum* is less than zero or greater than **sem_nsems**.

[EINVAL] *Cmd* is not a valid command.

[EACCES] Operation permission is denied to the calling process [see *intro*(2)].

[ERANGE] *Cmd* is **SETVAL** or **SETALL** and the value to which semval is to be set is greater than the system imposed maximum.

[EPERM] *Cmd* is equal to **IPC_RMID** or **IPC_SET** and the effective user ID of the calling process is not equal to that of super-user, or to the value of **sem_perm.cuid** or **sem_perm.uid** in the data structure associated with *semid*.

[EFAULT] *Arg.buf* points to an illegal address.

SEE ALSO

intro(2), semget(2), semop(2).

DIAGNOSTICS

Upon successful completion, the value returned depends on *cmd* as follows:

GETVAL	The value of semval.
GETPID	The value of sempid.
GETNCNT	The value of semncnt.
GETZCNT	The value of semzcnt.
All others	A value of 0.

Otherwise, a value of −1 is returned and *errno* is set to indicate the error.

NAME

semget – get set of semaphores

SYNOPSIS

#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/sem.h>

int semget (key_t key, int nsems, int semflg);

DESCRIPTION

semget returns the semaphore identifier associated with *key*.

A semaphore identifier and associated data structure and set containing *nsems* semaphores [see *intro*(2)] are created for *key* if one of the following is true:

Key is equal to **IPC_PRIVATE**.

Key does not already have a semaphore identifier associated with it, and (*semflg* & **IPC_CREAT**) is "true".

Upon creation, the data structure associated with the new semaphore identifier is initialized as follows:

Sem_perm.cuid, **sem_perm.uid**, **sem_perm.cgid**, and **sem_perm.gid** are set equal to the effective user ID and effective group ID, respectively, of the calling process.

The low-order 9 bits of **sem_perm.mode** are set equal to the low-order 9 bits of *semflg*.

Sem_nsems is set equal to the value of *nsems*.

Sem_otime is set equal to 0 and **sem_ctime** is set equal to the current time.

semget fails if one or more of the following are true:

[EINVAL] *Nsems* is either less than or equal to zero or greater than the system-imposed limit.

[EACCES] A semaphore identifier exists for *key*, but operation permission [see *intro*(2)] as specified by the low-order 9 bits of *semflg* would not be granted.

[EINVAL] A semaphore identifier exists for *key*, but the number of semaphores in the set associated with it is less than *nsems*, and *nsems* is not equal to zero.

[ENOENT]	A semaphore identifier does not exist for *key* and (*semflg* & **IPC_CREAT**) is ''false''.
[ENOSPC]	A semaphore identifier is to be created but the system-imposed limit on the maximum number of allowed semaphore identifiers system wide would be exceeded.
[ENOSPC]	A semaphore identifier is to be created but the system-imposed limit on the maximum number of allowed semaphores system wide would be exceeded.
[EEXIST]	A semaphore identifier exists for *key* but ((*semflg* & **IPC_CREAT**) and (*semflg* & **IPC_EXCL**)) is ''true''.

SEE ALSO

intro(2), semctl(2), semop(2).

DIAGNOSTICS

Upon successful completion, a non-negative integer, namely a semaphore identifier, is returned. Otherwise, a value of −1 is returned and *errno* is set to indicate the error.

NAME

semop – semaphore operations

SYNOPSIS

#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/sem.h>

int semop (int semid, struct sembuf *sops, unsigned nsops);

DESCRIPTION

semop is used to atomically perform an array of semaphore operations on the set of semaphores associated with the semaphore identifier specified by *semid*. *Sops* is a pointer to the array of semaphore-operation structures. *Nsops* is the number of such structures in the array. The contents of each structure includes the following members:

```
ushort   sem_num;   /* semaphore number */
short    sem_op;    /* semaphore operation */
short    sem_flg;   /* operation flags */
```

Each semaphore operation specified by *sem_op* is performed on the corresponding semaphore specified by *semid* and *sem_num*.

Sem_op specifies one of three semaphore operations as follows:

If *sem_op* is a negative integer, one of the following will occur: {ALTER}

If semval [see *intro*(2)] is greater than or equal to the absolute value of *sem_op*, the absolute value of *sem_op* is subtracted from semval. Also, if (*sem_flg* & **SEM_UNDO**) is "true", the absolute value of *sem_op* is added to the calling process's semadj value [see *exit*(2)] for the specified semaphore.

If semval is less than the absolute value of *sem_op* and (*sem_flg* & **IPC_NOWAIT**) is "true", *semop* will return immediately.

If semval is less than the absolute value of *sem_op* and (*sem_flg* & **IPC_NOWAIT**) is "false", *semop* will increment the semncnt associated with the specified semaphore and suspend execution of the calling process until one of the following conditions occur.

Semval becomes greater than or equal to the absolute value of *sem_op*. When this occurs, the value of semncnt associated with the specified semaphore is decremented, the absolute value of *sem_op* is subtracted

from semval and, if (*sem_flg* & **SEM_UNDO**) is "true", the absolute value of *sem_op* is added to the calling process's semadj value for the specified semaphore.

The semid for which the calling process is awaiting action is removed from the system [see *semctl*(2)]. When this occurs, *errno* is set equal to EIDRM, and a value of −1 is returned.

The calling process receives a signal that is to be caught. When this occurs, the value of semncnt associated with the specified semaphore is decremented, and the calling process resumes execution in the manner prescribed in *signal*(2).

If *sem_op* is a positive integer, the value of *sem_op* is added to semval and, if (*sem_flg* & **SEM_UNDO**) is "true", the value of *sem_op* is subtracted from the calling process's semadj value for the specified semaphore. {ALTER}

If *sem_op* is zero, one of the following will occur: {READ}

If semval is zero, *semop* will return immediately.

If semval is not equal to zero and (*sem_flg* & **IPC_NOWAIT**) is "true", *semop* will return immediately.

If semval is not equal to zero and (*sem_flg* & **IPC_NOWAIT**) is "false", *semop* will increment the semzcnt associated with the specified semaphore and suspend execution of the calling process until one of the following occurs:

Semval becomes zero, at which time the value of semzcnt associated with the specified semaphore is decremented.

The semid for which the calling process is awaiting action is removed from the system. When this occurs, *errno* is set equal to EIDRM, and a value of −1 is returned.

The calling process receives a signal that is to be caught. When this occurs, the value of semzcnt associated with the specified semaphore is decremented, and the calling process resumes execution in the manner prescribed in *signal*(2).

semop will fail if one or more of the following are true for any of the semaphore operations specified by *sops*:

[EINVAL] *Semid* is not a valid semaphore identifier.

[EFBIG] *Sem_num* is less than zero or greater than or equal to the number of semaphores in the set associated with *semid*.

[E2BIG] *Nsops* is greater than the system-imposed maximum.

[EACCES] Operation permission is denied to the calling process [see *intro*(2)]

[EAGAIN] The operation would result in suspension of the calling process but (*sem_flg* & **IPC_NOWAIT**) is "true".

[ENOSPC] The limit on the number of individual processes requesting an **SEM_UNDO** would be exceeded.

[EINVAL] The number of individual semaphores for which the calling process requests a **SEM_UNDO** would exceed the limit.

[ERANGE] An operation would cause a semval to overflow the system-imposed limit.

[ERANGE] An operation would cause a semadj value to overflow the system-imposed limit.

[EFAULT] *Sops* points to an illegal address.

Upon successful completion, the value of sempid for each semaphore specified in the array pointed to by *sops* is set equal to the process ID of the calling process.

SEE ALSO

exec(2), exit(2), fork(2), intro(2), semctl(2), semget(2).

DIAGNOSTICS

If *semop* returns due to the receipt of a signal, a value of −1 is returned to the calling process and *errno* is set to EINTR. If it returns due to the removal of a *semid* from the system, a value of −1 is returned and *errno* is set to EIDRM.

Upon successful completion, a value of zero is returned. Otherwise, a value of −1 is returned and *errno* is set to indicate the error.

NAME

send, sendto, sendmsg – send a message from a socket

C SYNOPSIS

```
#include <sys/types.h>
#include <sys/socket.h>

cc = send(s, msg, len, flags)
int cc, s;
char *msg;
int len, flags;

cc = sendto(s, msg, len, flags, to, tolen)
int cc, s;
char *msg;
int len, flags;
struct sockaddr *to;
int tolen;

cc = sendmsg(s, msg, flags)
int cc, s;
struct msghdr msg[];
int flags;
```

DESCRIPTION

Send, *sendto*, and *sendmsg* are used to transmit a message to another socket. *Send* may be used only when the socket is in a *connected* state, while *sendto* and *sendmsg* may be used at any time.

The address of the target is given by *to* with *tolen* specifying its size. The length of the message is given by *len*. If the message is too long to pass atomically through the underlying protocol, then the error EMSGSIZE is returned, and the message is not transmitted.

No indication of failure to deliver is implicit in a *send*. Return values of –1 indicate some locally detected errors.

If no messages space is available at the socket to hold the message to be transmitted, then *send* normally blocks, unless the socket has been placed in non-blocking I/O mode. The *select*(2) call may be used to determine when it is possible to send more data.

The *flags* parameter may include one or more of the following:

```
#define  MSG_OOB          0x1    /* process out-of-band data */
#define  MSG_DONTROUTE    0x4    /* bypass routing,
                                   use direct interface */
```

The flag MSG_OOB is used to send "out-of-band" data on sockets that support this notion (e.g., SOCK_STREAM); the underlying protocol must also support "out-of-band" data. MSG_DONTROUTE is usually used only by diagnostic or routing programs.

See *recv*(2) for a description of the *msghdr* structure.

RETURN VALUE

The call returns the number of characters sent, or −1 if an error occurred.

ERRORS

[EBADF]	An invalid descriptor was specified.
[ENOTSOCK]	The argument *s* is not a socket.
[EFAULT]	An invalid user space address was specified for a parameter.
[EMSGSIZE]	The socket requires that message be sent atomically, and the size of the message to be sent made this impossible.
[EWOULDBLOCK]	The socket is marked non-blocking and the requested operation would block.
[ENOBUFS]	The system was unable to allocate an internal buffer. The operation may succeed when buffers become available.
[ENOBUFS]	The output queue for a network interface was full. This generally indicates that the interface has stopped sending, but may be caused by transient congestion.

SEE ALSO

fcntl(2), recv(2), select(2), getsockopt(2), socket(2), write(2)

NAME

setgroups – set group access list

SYNOPSIS

#include <sys/param.h>

POSIX:

int setgroups(int ngroups, gid_t *gidset);

BSD:

int setgroups(int ngroups, int *gidset);

To use the BSD versions of *setgroups*, *getgroups*, or *initgroups*, you must either

1) explicitly invoke them as *BSDsetgroups*, *BSDgetgroups*, or *BSDinitgroups*, or

2) link with the libbsd.a library:

```
cc -o prog prog.c -lbsd
```

DESCRIPTION

setgroups sets the group access list of the current user process according to the array *gidset*. The parameter *ngroups* indicates the number of entries in the array and must be no more than **NGROUPS** (as defined in *<sys/param.h>*) for the BSD version. POSIX provides the *sysconf*(_SC_NGROUPS_MAX) system call to determine at runtime the maximum value for *ngroups* in the particular kernel configuration (it is an lbootable option which is set in /usr/sysgen/master.d/kernel).

Only the super-user may set new groups.

RETURN VALUE

A successful call returns the number of groups in the group set. A value of −1 indicates that an error occurred, and the error code is stored in the global variable *errno*.

ERRORS

The *setgroups* call will fail if:

[EPERM]	The caller is not the super-user.
[EFAULT]	The address specified for *gidset* is outside the process address space.
[EINVAL]	The *ngroups* parameter is greater than **NGROUPS**, as specified above.

SEE ALSO

multgrps(1), getgroups(2), initgroups(3), sysconf(2)

CAVEATS

The POSIX and 4.3BSD versions differ in the types of their *gidset* parameter.

NAME

setpgid – set process group ID

C SYNOPSIS

#include <sys/types.h>
#include <unistd.h>

int setpgid (pid_t pid, pid_t pgid);

DESCRIPTION

The *setpgid* function is used to either join an existing process group or create a new process group within the session of the calling process. The process group ID of a session leader is not allowed to change. Upon successful completion, the process group ID of the process with a process ID that matches *pid* shall be set to *pgid.* Special cases: if *pid* is zero, the process ID of the calling process is used, and if *pgid* is zero, the process ID of the indicated process is used.

setgpid will fail if one or more of the following are true:

[EACCES] The value of the *pid* argument matches the process ID of a child process of the calling process and the child process has successfully executed one of the *exec* functions.

[EINVAL] The value of the *pgid* argument is less than zero or is not a value supported by this implementation.

[EPERM] The process indicated by the *pid* argument is a session leader.

[EPERM] The value of the *pid* argument is valid but matches the process ID of a child process of the calling process, and the child is not in the same session as the calling process.

[EPERM] The value of the *pgid* argument does not match the process ID of the process indicated by the *pid* argument, and there is no process with a process group ID that matches the value of the *pgid* argument in the same session as the calling process.

[ESRCH] The value of the *pid* argument does not match the process ID of the calling process or of a child of the calling process.

SEE ALSO

getpgrp(2), setsid(2), setpgrp(2), exec(2), tcsetpgrp(3T).

DIAGNOSTICS

Upon successful completion, the *setpgid* function returns a value of zero. Otherwise, a value of -1 is returned and *errno* is set to indicate the error.

NAME

setpgrp, BSDsetpgrp – set process group ID (System V and 4.3BSD)

C SYNOPSIS

SysV:

int setpgrp (void);

BSD:

Links with the BSD version automatically:
int BSDsetpgrp(int pid, int pgrp);

Links with the BSD version if *–lbsd* is specified during link phase:
int setpgrp(int pid, int pgrp);

DESCRIPTION

The System V version of *setpgrp* sets the process group ID of the calling process to the process ID of the calling process and returns the new process group ID.

The BSD version of *setpgrp* set the process group of the specified process *pid* to the specified *pgrp*. If *pid* is zero, then the call applies to the current process.

If the invoker is not the super-user, then the affected process must have the same effective user-id as the invoker or be a descendant of the invoking process.

ERRORS: BSD VERSION ONLY

setpgrp and *BSDsetpgrp* will fail and the process group will not be altered if one of the following occur:

[ESRCH]	The requested process does not exist.
[EPERM]	The effective user ID of the requested process is different from that of the caller and the process is not a descendent of the calling process.

SEE ALSO

exec(2), fork(2), getpgrp(2), getpid(2), intro(2), kill(2), setpgid(2), signal(2).

DIAGNOSTICS

The System V version of *setpgrp* returns the value of the new process group ID with no possibility of error. The BSD version also returns the new process group ID if the operation was successful. If the request failed, –1 is returned and the global variable *errno* indicates the reason.

NAME

setregid – set real and effective group ID

SYNOPSIS

setregid(rgid, egid)
int rgid, egid;

DESCRIPTION

The real and effective group ID's of the current process are set to the arguments. Unprivileged users may change the real group ID to the effective group ID and vice-versa; only the super-user may make other changes.

Supplying a value of –1 for either the real or effective group ID forces the system to substitute the current ID in place of the –1 parameter.

RETURN VALUE

Upon successful completion, a value of 0 is returned. Otherwise, a value of –1 is returned and *errno* is set to indicate the error.

ERRORS

[EPERM] The current process is not the super-user and a change other than changing the effective group-id to the real group-id was specified.

SEE ALSO

getgid(2), setreuid(2), setgid(3)

NAME

setreuid – set real and effective user ID's

SYNOPSIS

setreuid(ruid, euid)
int ruid, euid;

DESCRIPTION

The real and effective user ID's of the current process are set according to the arguments. If *ruid* or *euid* is –1, the current uid is filled in by the system. Unprivileged users may change the real user ID to the effective user ID and vice-versa; only the super-user may make other changes.

RETURN VALUE

Upon successful completion, a value of 0 is returned. Otherwise, a value of –1 is returned and *errno* is set to indicate the error.

ERRORS

[EPERM] The current process is not the super-user and a change other than changing the effective user-id to the real user-id was specified.

SEE ALSO

getuid(2), setregid(2), setuid(3)

NAME

setsid – create session and set process group IDs

C SYNOPSIS

#include <sys/types.h>
#include <unistd.h>

pid_t setsid (void);

DESCRIPTION

If the calling process is not a process group leader, the *setsid* function creates a new session. The calling process is the session leader of this new session, the process group leader of a new process group, and has no controlling terminal. The process group ID of the calling process is set equal to the process ID of the calling process. The calling process is the only process in the new process group and the only process in the new session.

setsid will fail if the following is true:

[EPERM] The calling process is already a process group leader, or the process group ID of a process other than the calling process matches the process ID of the calling process.

DIAGNOSTICS

Upon successful completion, the *setsid* function returns the value of the process group ID of the calling process, else −1 and *errno* is set to the appropriate error.

SEE ALSO

exec(2), _exit(2), fork(2), getpid(2), kill(2), setpgid(2), sigaction(2).

NAME

setuid, setgid – set user and group IDs

C SYNOPSIS

#include <sys/types.h>
#include <unistd.h>

int setuid (uid_t uid);

int setgid (gid_t gid);

DESCRIPTION

setuid (*setgid*) is used to set the real user (group) ID and effective user (group) ID of the calling process.

If the effective user ID of the calling process is super-user, the real user (group) ID and effective user (group) ID are set to *uid* (*gid*).

If the effective user ID of the calling process is not super-user, but its real user (group) ID is equal to *uid* (*gid*), the effective user (group) ID is set to *uid* (*gid*).

If the effective user ID of the calling process is not super-user, but the saved set-user (group) ID from *exec*(2) is equal to *uid* (*gid*), the effective user (group) ID is set to *uid* (*gid*).

setuid or *setgid* will fail if one or more of the following are true:

[EPERM] *setuid* (*setgid*) will fail if the real user (group) ID of the calling process is not equal to *uid* (*gid*) and its effective user ID is not super-user.

[EINVAL] The *uid (gid)* is out of range.

SEE ALSO

getuid(2), intro(2).

DIAGNOSTICS

Upon successful completion, a value of 0 is returned. Otherwise, a value of −1 is returned and *errno* is set to indicate the error.

NAME

sgigsc – SGI graphics system call

SYNOPSIS

#include <sys/sgigsc.h>

int sgigsc(int op, ...);

DESCRIPTION

This system call is used for underlying operating system support of graphics and window management functions. It is not intended for direct use by user programs.

NAME

sgikopt – retrieve kernel option strings

C SYNOPSIS

int sgikopt (option, buf, buflen)
char *option;
char *buf;
int buflen;

DESCRIPTION

The IRIX kernel saves most of the IRIS PROM Monitor environment variables (option strings) for later use by the kernel as well as user programs. *sgikopt* allows the user to retrieve the values of these variables. The *option* argument specifies the name of the variable; the value of the option is returned in the buffer addressed by *buf*. The size in bytes of this buffer is given by *buflen*. The string returned by *sgikopt* is guaranteed to be null-terminated, even if the length of the option value string exceeds *buflen*.

The environment variables recognized by *sgikopt* are as follows:

bootfile	The name of the file to use for autobooting.
bootmode	The type of boot performed: **m** to enter the PROM Monitor after clearing memory, **d** to enter the PROM Monitor without clearing memory, or **c** to perform an autoboot using the bootfile.
console	The console to use: **g** for graphics console, **G** for graphics console with logo, or **d** for serial terminal port 1.
diskless	If this variable is set, the system expects to mount it's root filesystem via NFS from a server. It may be set even if a local disk is installed, in which case the local root disk is ignored.
dllogin	When *diskless* is set, this is the login name to use during installation on a diskless machine. See *cl_init*(1m) for more information.
dlserver	When *diskless* is set, this is the hostname of the machine from which the root filesystem will be mounted. It should be set in standard IP address format.
gfx	An variable indicating if graphics state: **alive** or **dead.**
hostname	The Internet host name string.

initfile	The process control initialization program if the default, */etc/init*, is overridden. See *init*(1M).
initstate	The run level to override the default state present in */etc/inittab*. See *inittab*(4).
keybd	The international keyboard type.
monitor	The graphics monitor type (*60*, *30*, *ntsc*, *pal*, or *343*) if the default (60HZ) is overridden.
netaddr	The Internet network address for booting across the Ethernet.
logocolor	The color of the logo while in the standalone programs, and the kernel textport (no window manager running) is controlled by this variable. It is 6 hex digits, which are the values for red, green, and blue in that order. This is currently implemented only on the Personal Iris.
nswap	The number of blocks to use in the swap partition; this amount overrides the partition size.
pagecolor	The color of the textport while in the standalone programs, and the kernel textport (no window manager running) is controlled by this variable. It is 6 hex digits, which are the values for red, green, and blue in that order. This is currently implemented only on the Personal Iris.
root	The disk that contains the root (/) file system (as it would be named in the /dev/dsk directory).
screencolor	The color of the screen background while in the standalone programs, and the kernel textport (no window manager running) is controlled by this variable. It is 6 hex digits, which are the values for red, green, and blue in that order. This is currently implemented only on the Personal Iris.
showconfig	If this variable is specified, the kernel will print out verbose information about memory and device configuration at boottime.
srvaddr	If set, this specifies the (only) host that will be used to resolve bootp requests. It should be set in standard IP address format.

swaplo	The first block to use in the swap partition; this amount overrides the default value of 0.
sync_on_green	If this value is **n**, the sync information is not superimposed on the green monitor line.
tapedevice	The tape device used for software installation.

DIAGNOSTICS

Upon successful completion, *sgikopt* returns 0. Otherwise, *sgikopt* returns −1 and sets *errno* to indicate the error:

[EINVAL]	The given *option* was not found.
[EFAULT]	The *option* or *buf* argument specifies an invalid address.

FILES

The list of variables that may be retrieved may be found in the file */usr/sysgen/master.d/kopt* for kernels built from those objects.

SEE ALSO

syssgi(2) - the **SGI_GETNVRAM** provides almost the same functionality.

NAME

sginap – timed sleep and processor yield function

C SYNOPSIS

void sginap (long ticks);

DESCRIPTION

The *sginap* system call provides two functions. With an argument of 0, it yields the processor to any higher or equal priority processes immediately, thus potentially allowing another process to run. Note that because normally the user has no direct control over the exact priority of a given process, this does not guarantee that another process will run.

With an argument which is non-zero, *sginap* will suspend the process for **ticks** clock ticks. The length of a clock tick is defined by **CLK_TCK** in the include file *<limits.h>*. This is the same for all *IRIS-4D* products.

SEE ALSO

sleep(3), alarm(2), pause(2), schedctl(2), setitimer(2).

NAME

shmctl – shared memory control operations

SYNOPSIS

#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>

int shmctl (int shmid, int cmd, ...);

Type of optional third argument:
struct shmid_ds *buf;

DESCRIPTION

shmctl provides a variety of shared memory control operations as specified by *cmd*. The following *cmd*s are available:

IPC_STAT Place the current value of each member of the data structure associated with *shmid* into the structure pointed to by *buf*. The contents of this structure are defined in *intro*(2). {READ}

IPC_SET Set the value of the following members of the data structure associated with *shmid* to the corresponding value found in the structure pointed to by *buf*:

shm_perm.uid
shm_perm.gid
shm_perm.mode /* only low 9 bits */

This *cmd* can only be executed by a process that has an effective user ID equal to that of super user, or to the value of **shm_perm.cuid** or **shm_perm.uid** in the data structure associated with *shmid*.

IPC_RMID Remove the shared memory identifier specified by *shmid* from the system and destroy the shared memory segment and data structure associated with it. This *cmd* can only be executed by a process that has an effective user ID equal to that of super user, or to the value of **shm_perm.cuid** or **shm_perm.uid** in the data structure associated with *shmid*.

SHM_LOCK

Lock the shared memory segment specified by *shmid* in memory. This *cmd* can only be executed by a process that has an effective user ID equal to super user.

SHM_UNLOCK
Unlock the shared memory segment specified by *shmid*. This *cmd* can only be executed by a process that has an effective user ID equal to super user.

shmctl will fail if one or more of the following are true:

[EINVAL] *Shmid* is not a valid shared memory identifier.

[EINVAL] *Cmd* is not a valid command.

[EINVAL] *Cmd* is SHM_LOCK and *shmid* corresponds to a shared memory segment for which a *shmat*(2) operation has not yet occurred.

[EACCES] *Cmd* is equal to **IPC_STAT** and {READ} operation permission is denied to the calling process [see *intro*(2)].

[EPERM] *Cmd* is equal to **IPC_RMID** or **IPC_SET** and the effective user ID of the calling process is not equal to that of super user, or to the value of **shm_perm.cuid** or **shm_perm.uid** in the data structure associated with *shmid*.

[EPERM] *Cmd* is equal to **SHM_LOCK** or **SHM_UNLOCK** and the effective user ID of the calling process is not equal to that of super user.

[EFAULT] *Buf* points to an illegal address.

[ENOMEM] *Cmd* is equal to **SHM_LOCK** and there is not enough memory.

SEE ALSO
shmget(2), shmop(2).

DIAGNOSTICS
Upon successful completion, a value of 0 is returned. Otherwise, a value of −1 is returned and *errno* is set to indicate the error.

NOTES
The user must explicitly remove shared memory segments after the last reference to them has been removed.

NAME

shmget – get shared memory segment identifier

SYNOPSIS

#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>

int shmget (key_t key, int size, int shmflg);

DESCRIPTION

shmget returns the shared memory identifier associated with *key*.

A shared memory identifier and associated data structure and shared memory segment of at least *size* bytes [see *intro*(2)] are created for *key* if one of the following are true:

Key is equal to **IPC_PRIVATE**.

Key does not already have a shared memory identifier associated with it, and (*shmflg* & **IPC_CREAT**) is "true".

Upon creation, the data structure associated with the new shared memory identifier is initialized as follows:

Shm_perm.cuid, **shm_perm.uid**, **shm_perm.cgid**, and **shm_perm.gid** are set equal to the effective user ID and effective group ID, respectively, of the calling process.

The low-order 9 bits of **shm_perm.mode** are set equal to the low-order 9 bits of *shmflg*. **Shm_segsz** is set equal to the value of *size*.

Shm_lpid, **shm_nattch**, **shm_atime**, and **shm_dtime** are set equal to 0.

Shm_ctime is set equal to the current time.

Note that if no permission bits are set into *shmflg*, the newly created segment will be accessible by the super-user.

shmget will fail if one or more of the following are true:

[EINVAL] *Size* is less than the system-imposed minimum or greater than the system-imposed maximum *{SHMMAX}* [see *intro*(2)].

[EACCES] A shared memory identifier exists for *key* but operation permission [see *intro*(2)] as specified by the low-order 9 bits of *shmflg* would not be granted.

[EINVAL] A shared memory identifier exists for *key* but the size of the segment associated with it is less than *size* and *size* is not equal to zero.

[ENOENT] A shared memory identifier does not exist for *key* and (*shmflg* & **IPC_CREAT**) is ''false''.

[ENOSPC] A shared memory identifier is to be created but the system-imposed limit on the maximum number of allowed shared memory identifiers system wide *{SHMSEG_MAX}* [see *intro*(2)] would be exceeded.

[EEXIST] A shared memory identifier exists for *key* but ((*shmflg* & **IPC_CREAT**) and (*shmflg* & **IPC_EXCL**)) is ''true''.

SEE ALSO

intro(2), shmctl(2), shmop(2).

DIAGNOSTICS

Upon successful completion, a non-negative integer, namely a shared memory identifier is returned. Otherwise, a value of −1 is returned and *errno* is set to indicate the error.

NOTES

The user must explicitly remove shared memory segments after the last reference to them has been removed.

NAME

shmop: shmat, shmdt – shared memory operations

C SYNOPSIS

#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>

void *shmat (int shmid, void *shmaddr, int shmflg);

int shmdt (void *shmaddr);

DESCRIPTION

shmat attaches the shared memory segment associated with the shared memory identifier specified by *shmid* to the data segment of the calling process. The segment is attached at the address specified by one of the following criteria:

If *shmaddr* is equal to zero, the segment is attached at the first available address as selected by the system.

If *shmaddr* is not equal to zero and (*shmflg* & **SHM_RND**) is ''true'', the segment is attached at the address given by (*shmaddr* - (*shmaddr* modulus **SHMLBA**)).

If *shmaddr* is not equal to zero and (*shmflg* & **SHM_RND**) is ''false'', the segment is attached at the address given by *shmaddr*.

shmdt detaches from the calling process's data segment the shared memory segment located at the address specified by *shmaddr*.

The segment is attached for reading if (*shmflg* & **SHM_RDONLY**) is ''true'' {READ}, otherwise it is attached for reading and writing {READ/WRITE}.

shmat will fail and not attach the shared memory segment if one or more of the following are true:

[EINVAL] *Shmid* is not a valid shared memory identifier.

[EACCES] Operation permission is denied to the calling process [see *intro*(2)].

[ENOMEM] The available virtual space of the caller (either total size *{PROCSIZE_MAX}* or a large enough gap between other previously allocated virtual spaces) cannot accommodate the shared memory segment.

[EINVAL] *Shmaddr* is not equal to zero, and the value of (*shmaddr* - (*shmaddr* modulus **SHMLBA**)) is an illegal address.

[EINVAL] *Shmaddr* is not equal to zero, (*shmflg* & **SHM_RND**) is "false", and the value of *shmaddr* is an illegal address. Attach addresses must be a multiple of **SHMLBA** [see **<sys/shm.h>**].

[EMFILE] The number of shared memory segments attached to the calling process would exceed the system-imposed limit *{SHMAT_MAX}* [see *intro*(2)].

shmdt will fail and not detach the shared memory segment if one or more of the following are true:

[EBUSY] The shared memory segment is in use by another member of the calling process's share group [see *sproc*(2)].

[EINVAL] *Shmaddr* is not the start address of a shared memory segment.

SEE ALSO

exec(2), exit(2), fork(2), intro(2), shmctl(2), shmget(2), sproc(2).

DIAGNOSTICS

Upon successful completion, the return value is as follows:

shmat returns the data segment start address of the attached shared memory segment.

shmdt returns a value of 0.

Otherwise, a value of −1 is returned and *errno* is set to indicate the error.

NOTES

The user must explicitly remove shared memory segments after the last reference to them has been removed.

NAME

shutdown – shut down part of a full-duplex connection

SYNOPSIS

shutdown(s, how)
int s, how;

DESCRIPTION

The *shutdown* call causes all or part of a full-duplex connection on the socket associated with *s* to be shut down. If *how* is 0, then further receives will be disallowed. If *how* is 1, then further sends will be disallowed. If *how* is 2, then further sends and receives will be disallowed.

DIAGNOSTICS

A 0 is returned if the call succeeds, –1 if it fails.

ERRORS

The call succeeds unless:

[EBADF]	*S* is not a valid descriptor.
[ENOTSOCK]	*S* is a file, not a socket.
[ENOTCONN]	The specified socket is not connected.

SEE ALSO

connect(2), socket(2)

NAME

sigaction – software signal facilities (POSIX)

SYNOPSIS

```
#include <signal.h>

struct sigaction {
        void (*sa_handler)(int, ...);
        sigset_t sa_mask;
        int sa_flags;
};

int sigaction(int sig, struct sigaction *act, struct sigaction *oact);
```

DESCRIPTION

sigaction specifies and reports on the way individual signals are to be handled in the calling process. If *act* is non-zero, it alters the way the signal will be treated—default behavior, ignored, or handled via a routine—and the signal mask to be used when delivering the signal if a handler is installed. If *oact* is non-zero, the previous handling information for the signal is returned to the user. In this way (a NULL *act* and a non-NULL *oact)* the user can enquire as to the current handling of a signal without changing it. If both *act* and *oact* are NULL, *sigaction* will return **EINVAL** if *sig* is an invalid signal (else 0), allowing an application to dynamically determine the set of signals supported by the system.

If a handler is specified in the *sa_handler* field, the signals contained in *sa_mask* plus the signal being caught will be *added to* the process's signal mask before the signal-catching function is invoked. If *sa_handler* is **SIG_DFL** or **SIG_IGN** the *sa_mask* field is ignored. The user constructs this mask via the routines described in *sigsetops*(3).

The *sa_flags* field has only one value intended for user consumption: if **SA_NOCLDSTOP** (defined in <*sys/signal.h*>) is set by a parent process via a *sigaction* call with *sig* equal to **SIGCHLD**, that process will NOT receive a **SIGCHLD** signal when any of its child-processes stop. (By default parent processes *DO* receive this signal.) The **_SA_BSDCALL** value is for use by BSD compatible signal routines only. Since the default behavior is generally desired, most users will set the *sa_flags* field to 0 before calling *sigaction.*

The system defines a set of signals that may be delivered to a process. Signal delivery resembles the occurrence of a hardware interrupt: the current process context is saved, and a new one is built. A process may specify 1) a *handler* to which a signal is delivered and a signal mask to install while the handler executes, 2) that a signal is to be *ignored*, or 3) that a default action is to be taken by the system when a signal occurs.

All signals have the same *priority*. A global *signal mask* defines the set of signals currently blocked from delivery to a process. The signal mask for a process is initialized from that of its parent (normally 0). It may be changed with a *sigprocmask*(2) call, or when a signal is delivered to the process.

Routines described in *sigsetops*(3) are used to create and manipulate the input-parameter signal masks submitted to *sigaction*(2), *sigprocmask*(2), and *sigsuspend*(2), and returned by *sigpending*(2). These masks are of type *sigset_t*.

When a signal condition arises for a process, the signal is added to a set of signals pending for the process. If the signal is not currently *blocked* by the process then it is delivered to the process. When a signal is delivered, the current state of the process is saved, a new signal mask is calculated (as described below), and the signal handler is invoked. The call to the handler is arranged so that if the signal handling routine returns normally the process will resume execution in the context from before the signal's delivery. If the process wishes to resume in a different context, then it must arrange to restore the previous context itself.

When a signal is delivered to a process a new signal mask is installed for the duration of the process' signal handler (or until a *sigprocmask* call is made). This mask is formed by taking the current signal mask, adding the signal to be delivered, and *or*'ing in the signal mask associated with the handler to be invoked.

Once a signal handler is installed, it remains installed until another *sigaction* call is made, or an *execve*(2) is performed. The default action for a signal may be reinstated by setting *sa_handler* to **SIG_DFL**; this default is termination with a core image for signals marked [1].

The following is a list of all signals with names as in the include file *<sys/signal.h>*:

SIGHUP	01	hangup
SIGINT	02	interrupt
SIGQUIT	03[1]	quit
SIGILL	04[1]	illegal instruction (not reset when caught)
SIGTRAP	05[1][5]	trace trap (not reset when caught)
SIGABRT	06[1]	abort
SIGEMT	07[1][4]	EMT instruction
SIGFPE	08[1]	floating point exception
SIGKILL	09	kill (cannot be caught or ignored)
SIGBUS	10[1]	bus error
SIGSEGV	11[1]	segmentation violation

SIGSYS	12[1]	bad argument to system call
SIGPIPE	13	write on a pipe with no one to read it
SIGALRM	14	alarm clock
SIGTERM	15	software termination signal
SIGUSR1	16	user-defined signal 1
SIGUSR2	17	user-defined signal 2
SIGCLD	18[2]	death of a child
SIGPWR	19[2]	power fail (not reset when caught)
SIGSTOP	20[6]	stop (cannot be caught or ignored)
SIGTSTP	21[6]	stop signal generated from keyboard
SIGPOLL	22[3]	selectable event pending
SIGIO	23[2]	input/output possible
SIGURG	24[2]	urgent condition on IO channel
SIGWINCH	25[2]	window size changes
SIGVTALRM	26	virtual time alarm
SIGPROF	27	profiling alarm
SIGCONT	28[6]	continue after stop (cannot be ignored)
SIGTTIN	29[6]	background read from control terminal
SIGTTOU	30[6]	background write to control terminal
SIGXCPU	31	cpu time limit exceeded [see *setrlimit*(2)]
SIGXFSZ	32	file size limit exceeded [see *setrlimit*(2)]

sa_handler is assigned one of three values: **SIG_DFL** or **SIG_IGN**, which are macros (defined in <*sys/signal.h*>) that expand to constant expressions, or a *function address*.

The actions prescribed by its value are as follows:

SIG_DFL – terminate process upon receipt of a signal

Upon receipt of the signal *sig*, the receiving process is to be terminated with all of the consequences outlined in *exit*(2). See SIGNAL NOTES [1] below.

SIG_IGN – ignore signal

The signal *sig* is to be ignored.

Note: the signals **SIGKILL, SIGSTOP** and **SIGCONT** cannot be ignored.

function address – catch signal

Upon receipt of the signal *sig*, the receiving process is to execute the signal-catching function whose address is specified via this parameter. The function will be invoked as follows:

handler (int sig, int code, struct sigcontext *sc);

Where *handler* is the specified function-name. *code* is valid only in the following cases:

Condition	Signal	Code
User breakpoint	SIGTRAP	BRK_USERBP
User breakpoint	SIGTRAP	BRK_SSTEPBP
Integer overflow	SIGTRAP	BRK_OVERFLOW
Divide by zero	SIGTRAP	BRK_DIVZERO
Multiply overflow	SIGTRAP	BRK_MULOVF
Invalid virtual address	SIGSEGV	EFAULT
Read-only address	SIGSEGV	EACCESS
Read beyond mapped object	SIGSEGV	ENXIO

The third argument *sc* is a pointer to a *struct sigcontext* (defined in <*sys/signal.h*>) that contains the processor context at the time of the signal.

The signal-catching function remains installed after it is invoked. During normal return from the signal-catching handler, the system signal action is restored to *func* and any held signal of this type released. If a non-local goto (*longjmp*) is taken, then *sigrelse* must be called to restore the system signal action and release any held signal of this type.

Upon return from the signal-catching function, the receiving process will resume execution at the point it was interrupted. See WARNINGS below.

When a signal that is to be caught occurs during a *read*(2), a *write*(2), an *open*(2), or an *ioctl*(2) system call on a slow device (like a terminal; but not a file), during a *pause*(2) system call, or during a *wait*(2) system call that does not return immediately due to the existence of a previously stopped or zombie process, the signal catching function will be executed and then the interrupted system call may return a –1 to the calling process with *errno* set to **EINTR**.

Note: The signals **SIGKILL** and **SIGSTOP** cannot be caught.

SIGNAL NOTES

[1] If **SIG_DFL** is assigned for these signals, in addition to the process being terminated, a "core image" will be constructed in the current working directory of the process, if the following conditions are met:

The effective user ID and the real user ID of the receiving process are equal.

An ordinary file named **core** exists and is writable or can be created. If the file must be created, it will have the following properties:

a mode of 0666 modified by the file creation mask [see *umask*(2)]

a file owner ID that is the same as the effective user ID of the receiving process.

a file group ID that is the same as the effective group ID of the receiving process

NOTE: The core file may be truncated if the resultant file size would exceed either *ulimit* [see *ulimit*(2)] or the process's maximum core file size [see *setrlimit*(2)].

[2] For the signals **SIGCLD**, **SIGWINCH**, **SIGPWR**, **SIGURG**, and **SIGIO**, the handler parameter is assigned one of three values: **SIG_DFL**, **SIG_IGN**, or a *function address*. The actions prescribed by these values are:

SIG_DFL – ignore signal
The signal is to be ignored.

SIG_IGN – ignore signal
The signal is to be ignored.

function address – catch signal
If the signal is **SIGPWR**, **SIGWINCH**, **SIGURG**, or **SIGIO**, the action to be taken is the same as that described above for a handler parameter equal to *function address*. The same is true if the signal is **SIGCLD** with one exception: while the process is executing the signal-catching function, all terminating child processes will be queued. The *wait* system call removes the first entry of the queue. Unlike *signal*(2) and *sigset*(2), re-attaching the handler before exiting it will NOT ensure that no **SIGCLD**'s will be missed. See *wait*(2) for an example of parent code waiting on children.

When processing a pipeline, the shell makes the last process in the pipeline the parent of the proceeding processes. A process that may be piped into in this manner (and thus become the parent of other processes) should take care not to set **SIGCLD** to be caught.

[3] **SIGPOLL** is issued when a file descriptor corresponding to a STREAMS [see *intro*(2)] file has a "selectable" event pending. A process must specifically request that this signal be sent using the **I_SETSIG** *ioctl* call. Otherwise, the process will never receive **SIGPOLL**.

[4] **SIGEMT** is never generated on an **IRIS-4D** system.

[5] **SIGTRAP** is generated for breakpoint instructions, overflows, divide by zeros, range errors, and multiply overflows. The second argument *code* gives specific details of the cause of the signal. Possible values are described in <*sys/signal.h*>.

[6] The signals **SIGSTOP**, **SIGTSTP**, **SIGTTIN**, **SIGTTOU** and **SIGCONT** are used by command interpreters like the C shell [see *csh*(1)] to provide job control. The first four signals listed will cause the receiving process to be stopped, unless the signal is caught or ignored. **SIGCONT** causes a stopped process to be resumed. **SIGTSTP** is sent from the terminal driver in response to the SWTCH character being entered from the keyboard [see *termio*(7)]. **SIGTTIN** is sent from the terminal driver when a background process attempts to read from its controlling terminal. If **SIGTTIN** is ignored by the process, then the read will return **EIO**. **SIGTTOU** is sent from the terminal driver when a background process attempts to write to its controlling terminal when the terminal is in **TOSTOP** mode. If **SIGTTOU** is ignored by the process, then the write will succeed regardless of the state of the controlling terminal.

NOTES

The mask specified in *act* is not allowed to block **SIGKILL** or **SIGSTOP**. This is done silently by the system.

SIGKILL will immediately terminate a process, regardless of its state. Processes which are stopped via job control (typically <Ctrl>-Z) will not act upon any delivered signals other than **SIGKILL** until the job is restarted. Processes which are blocked via a *blockproc* system call will unblock if they receive a signal which is fatal (i.e., a non-job-control signal which the are NOT catching), but will still be stopped if the job of which they are a part is stopped. Only upon restart will they die. Any non-fatal signals received by a blocked process will NOT cause the process to be unblocked (an *unblockproc*(2) or *unblockprocall*(2) system call is necessary).

After a *fork*(2) the child inherits all handlers and signal masks, but not the set of pending signals.

The *exec*(2) routines reset all caught signals to default action and clear all handler masks. Ignored signals remain ignored; the blocked signal mask is unchanged and pending signals remain pending.

POSIX specifies (contrary to BSD and SysV) that a process *may* block **SIGCONT**. However, a) **SIGCONT** *always* restarts the receiving process (unless it is waiting for an event such as I/O), and b) if the receiving process has installed a handler for **SIGCONT** and blocked the signal, the process will NOT enter its handler until it unblocks **SIGCONT**. (The signal will remain pending.)

sigaction will fail and no new signal handler will be installed if one of the following occurs:

[EFAULT] Either *act* or *oact* points to memory that is not a valid part of the process address space.

[EINVAL] *Sig* is not a valid signal number.

[EINVAL] An attempt is made to ignore or supply a handler for **SIGKILL** or **SIGSTOP**.

[EINVAL] An attempt is made to ignore **SIGCONT** (by default **SIGCONT** is ignored).

SEE ALSO

blockproc(2), kill(2), signal(2), sigprocmask(2), sigpending(2), sigsuspend(2), sigsetjmp(3), sigset(2), setrlimit(2), ulimit(2), wait(2), sigsetops(3), sigvec(3B).

DIAGNOSTICS

A 0 value indicates that the call succeeded. A –1 return value indicates an error occurred and *errno* is set to indicate the reason.

WARNINGS

Signals raised by the instruction stream, **SIGILL**, **SIGEMT**, **SIGBUS**, **SIGSEGV** will cause infinite loops if their handler returns, or the action is set to **SIG_IGN**.

WARNING

The POSIX signal routines (*sigaction*(2), *sigpending*(2), *sigprocmask*(2), *sigsuspend*(2), *sigsetjmp*(3)), and the 4.3BSD signal routines (*sigvec*(3B), *signal*(3B), *sigblock*(3B), *sigpause*(3B), *sigsetmask*(3B)) must NEVER be used with *signal*(2) or *sigset*(2).

NAME

signal – software signal facilities (System V)

C SYNOPSIS

#include <signal.h>

void (*signal (int sig, void (*func)(int, ...)))(int, ...);

DESCRIPTION

signal allows the calling process to choose one of three ways in which it is possible to handle the receipt of a specific signal. *Sig* specifies the signal and *func* specifies the choice. *Sig* is the signal to be caught, and must be in the range

(0 < sig < NSIG).

Sig can be assigned any one of the following except **SIGKILL** or **SIGSTOP**:

SIGHUP	01	hangup
SIGINT	02	interrupt
SIGQUIT	03[1]	quit
SIGILL	04[1]	illegal instruction (not reset when caught)
SIGTRAP	05[1][5]	trace trap (not reset when caught)
SIGABRT	06[1]	abort
SIGEMT	07[1][4]	EMT instruction
SIGFPE	08[1]	floating point exception
SIGKILL	09	kill (cannot be caught or ignored)
SIGBUS	10[1]	bus error
SIGSEGV	11[1]	segmentation violation
SIGSYS	12[1]	bad argument to system call
SIGPIPE	13	write on a pipe with no one to read it
SIGALRM	14	alarm clock
SIGTERM	15	software termination signal
SIGUSR1	16	user-defined signal 1
SIGUSR2	17	user-defined signal 2
SIGCLD	18[2]	death of a child
SIGPWR	19[2]	power fail (not reset when caught)
SIGSTOP	20[6]	stop (cannot be caught or ignored)
SIGTSTP	21[6]	stop signal generated from keyboard
SIGPOLL	22[3]	selectable event pending
SIGIO	23[2]	input/output possible
SIGURG	24[2]	urgent condition on IO channel
SIGWINCH	25[2]	window size changes
SIGVTALRM	26	virtual time alarm
SIGPROF	27	profiling alarm

SIGCONT	28[6]	continue after stop (cannot be ignored)
SIGTTIN	29[6]	background read from control terminal
SIGTTOU	30[6]	background write to control terminal
SIGXCPU	31	cpu time limit exceeded [see *setrlimit*(2)]
SIGXFSZ	32	file size limit exceeded [see *setrlimit*(2)]

Func is assigned one of three values: **SIG_DFL** or **SIG_IGN**, which are macros (defined in <*sys/signal.h*>) that expand to constant expressions, or a *function address*.

The actions prescribed by its value are as follows:

SIG_DFL – terminate process upon receipt of a signal

Upon receipt of the signal *sig*, the receiving process is to be terminated with all of the consequences outlined in *exit*(2). See SIGNAL NOTES [1] below.

SIG_IGN – ignore signal

The signal *sig* is to be ignored.

Note: the signals **SIGKILL, SIGSTOP** and **SIGCONT** cannot be ignored.

function address – catch signal

Upon receipt of the signal *sig*, the receiving process is to execute the signal-catching function whose address is specified via this parameter. The function will be invoked as follows:

handler (int sig, int code, struct sigcontext *sc);

Where *handler* is the specified function-name. *code* is valid only in the following cases:

Condition	Signal	Code
User breakpoint	SIGTRAP	BRK_USERBP
User breakpoint	SIGTRAP	BRK_SSTEPBP
Integer overflow	SIGTRAP	BRK_OVERFLOW
Divide by zero	SIGTRAP	BRK_DIVZERO
Multiply overflow	SIGTRAP	BRK_MULOVF
Invalid virtual address	SIGSEGV	EFAULT
Read-only address	SIGSEGV	EACCESS
Read beyond mapped object	SIGSEGV	ENXIO

The third argument *sc* is a pointer to a *struct sigcontext (defined in <sys/signal.h>)* that contains the processor context at the time of the signal.

Upon return from the signal-catching function, the receiving process will resume execution at the point it was interrupted.

Before entering the signal-catching function, the value of *func* for the caught signal will be set to **SIG_DFL** unless the signal is **SIGILL, SIGTRAP,** or **SIGPWR**. This means that before exiting the handler, a *signal* call is necessary to again set the disposition to catch the signal.

When a signal that is to be caught occurs during a *read*(2), a *write*(2), an *open*(2), or an *ioctl*(2) system call on a slow device (like a terminal; but not a file), during a *pause*(2) system call, or during a *wait*(2) system call that does not return immediately due to the existence of a previously stopped or zombie process, the signal catching function will be executed and then the interrupted system call may return a −1 to the calling process with *errno* set to **EINTR**.

Note: The signals **SIGKILL** and **SIGSTOP** cannot be caught.

SIGNAL NOTES

[1] If **SIG_DFL** is assigned for these signals, in addition to the process being terminated, a "core image" will be constructed in the current working directory of the process, if the following conditions are met:

The effective user ID and the real user ID of the receiving process are equal.

An ordinary file named **core** exists and is writable or can be created. If the file must be created, it will have the following properties:

a mode of 0666 modified by the file creation mask [see *umask*(2)]

a file owner ID that is the same as the effective user ID of the receiving process.

a file group ID that is the same as the effective group ID of the receiving process

NOTE: The core file may be truncated if the resultant file size would exceed either *ulimit* [see *ulimit*(2)] or the process's maximum core file size [see *setrlimit*(2)].

[2] For the signals **SIGCLD, SIGWINCH, SIGPWR, SIGURG,** and **SIGIO**, the handler parameter is assigned one of three values: **SIG_DFL, SIG_IGN,** or a *function address*. The actions prescribed by these values are:

SIG_DFL – ignore signal
The signal is to be ignored.

SIG_IGN – ignore signal
The signal is to be ignored. Also, if *sig* is **SIGCLD**, the calling process's child processes will not create zombie processes when they terminate [see *exit*(2)].

function address – catch signal
If the signal is **SIGPWR**, **SIGURG**, **SIGIO**, or **SIGWINCH**, the action to be taken is the same as that described above for a handler parameter equal to *function address*. The same is true if the signal is **SIGCLD** with one exception: while the process is executing the signal-catching function, all terminating child processes will be queued. The *wait* system call removes the first entry of the queue. If the *signal* system call is used to catch **SIGCLD**, the signal handler must be re-attached when exiting the handler, and at that time--if the queue is not empty--**SIGCLD** is re-raised before *signal* returns. See *wait*(2).

In addition, **SIGCLD** affects the *wait* and *exit* system calls as follows:

wait If the handler parameter of **SIGCLD** is set to **SIG_IGN** and a *wait* is executed, the *wait* will block until all of the calling process's child processes terminate; it will then return a value of –1 with *errno* set to **ECHILD**.

exit If in the exiting process's parent process the handler parameter of **SIGCLD** is set to **SIG_IGN**, the exiting process will not create a zombie process.

When processing a pipeline, the shell makes the last process in the pipeline the parent of the proceeding processes. A process that may be piped into in this manner (and thus become the parent of other processes) should take care not to set **SIGCLD** to be caught.

[3] **SIGPOLL** is issued when a file descriptor corresponding to a STREAMS [see *intro*(2)] file has a "selectable" event pending. A process must specifically request that this signal be sent using the **I_SETSIG** *ioctl* call. Otherwise, the process will never receive **SIGPOLL**.

[4] **SIGEMT** is never generated on an **IRIS-4D** system.

[5] **SIGTRAP** is generated for breakpoint instructions, overflows, divide by zeros, range errors, and multiply overflows. The second argument *code* gives specific details of the cause of the signal. Possible values are described in <*sys/signal.h*>.

[6] The signals **SIGSTOP, SIGTSTP, SIGTTIN, SIGTTOU** and **SIGCONT** are used by command interpreters like the C shell [see *csh*(1)] to provide job control. The first four signals listed will cause the receiving process to be stopped, unless the signal is caught or ignored. **SIGCONT** causes a stopped process to be resumed. **SIGTSTP** is sent from the terminal driver in response to the SWTCH character being entered from the keyboard [see *termio*(7)]. **SIGTTIN** is sent from the terminal driver when a background process attempts to read from its controlling terminal. If **SIGTTIN** is ignored by the process, then the read will return **EIO**. **SIGTTOU** is sent from the terminal driver when a background process attempts to write to its controlling terminal when the terminal is in **TOSTOP** mode. If **SIGTTOU** is ignored by the process, then the write will succeed regardless of the state of the controlling terminal.

NOTES

signal will not catch an invalid function argument, *func*, and results are undefined when an attempt is made to execute the function at the bad address.

SIGKILL will immediately terminate a process, regardless of its state. Processes which are stopped via job control (typically <Ctrl>-Z) will not act upon any delivered signals other than **SIGKILL** until the job is restarted. Processes which are blocked via a *blockproc* system call will unblock if they receive a signal which is fatal (i.e., a non-job-control signal which the are NOT catching), but will still be stopped if the job of which they are a part is stopped. Only upon restart will they die. Any non-fatal signals received by a blocked process will NOT cause the process to be unblocked (an *unblockproc*(2) or *unblockprocall*(2) system call is necessary).

A call to *signal* cancels a pending signal *sig* except for a pending **SIGKILL** signal.

[EINVAL] *signal* will fail if *sig* is an illegal signal number, including **SIGKILL** and **SIGSTOP**.

[EINVAL] *signal* will fail if an illegal operation is requested (for example, ignoring **SIGCONT**, which is ignored by default).

After a *fork*(2) the child inherits all handlers and signal masks, but not the set of the pending signals.

The *exec*(2) routines reset all caught signals to the default action; ignored signals remain ignored; the blocked signal mask is unchanged and pending signals remain pending.

SEE ALSO

intro(2), blockproc(2), kill(2), pause(2), ptrace(2), sigaction(2), sigset(2), wait(2), setjmp(3C), sigvec(3B).
kill(1) in the *User's Reference Manual.*

DIAGNOSTICS

Upon successful completion, *signal* returns the previous value of *func* for the specified signal *sig*. Otherwise, a value of **SIG_ERR** is returned and *errno* is set to indicate the error. **SIG_ERR** is defined in the header file *<sys/signal.h>*.

WARNINGS

Signals raised by the instruction stream, **SIGILL, SIGEMT, SIGBUS, SIGSEGV** will cause infinite loops if their handler returns, or the action is set to **SIG_IGN.**

WARNING

The POSIX signal routines (*sigaction*(2), *sigpending*(2), *sigprocmask*(2), *sigsuspend*(2), *sigsetjmp*(3)), and the 4.3BSD signal routines (*sigvec*(3B), *signal*(3B), *sigblock*(3B), *sigpause*(3B), *sigsetmask*(3B)) must NEVER be used with *signal*(2) or *sigset*(2).

Before entering the signal-catching function, the value of *func* for the caught signal will be set to **SIG_DFL** unless the signal is **SIGILL, SIGTRAP,** or **SIGPWR.** This means that before exiting the handler, a *signal* call is necessary to again set the disposition to catch the signal.

Note that handlers installed by *signal* execute with *no* signals blocked, not even the one that invoked the handler.

NAME

sigpending – return set of signals pending for process (POSIX)

SYNOPSIS

#include <signal.h>

int sigpending (sigset_t *maskptr);

DESCRIPTION

sigpending returns the set of signals pending for the calling process (i.e., blocked from delivery) in the space pointed to by *maskptr*.

Routines described in *sigsetops*(3) are used to examine the returned signal set.

sigpending will fail if:

[EFAULT] *maskptr* points to memory that is not a part of process's valid address space.

SEE ALSO

kill(2), sigaction(2), sigprocmask(2), sigsuspend(2), sigsetops(3).

DIAGNOSTICS

A 0 value indicates that the call succeeded. A –1 return value indicates an error occurred and *errno* is set to indicate the reason.

WARNING

The POSIX and System V signal facilities have different semantics. Using both facilities in the same program is **strongly discouraged** and will result in unpredictable behavior.

NAME

sigprocmask – alter and return previous state of the set of blocked signals (POSIX)

SYNOPSIS

#include <signal.h>

int sigprocmask(int operation, sigset_t *set, sigset_t *oset);

DESCRIPTION

sigprocmask manipulates the set of signals which are blocked from delivery to the process.

A non-NULL *set* specifies the set of signals to use in modifying the currently-active set, and the incoming signals may be added to, deleted from, or completely replace the active set, as specified by the *operation* parameter, which may have the following values (as defined in *<signal.h>*):

SIG_NOP	Do not alter current signal mask
SIG_BLOCK	Add specified signals to those in current mask
SIG_UNBLOCK	Remove the specified signals from current mask
SIG_SETMASK	Replace current mask with incoming one

If *oset* is not NULL, the current set of blocked signals (before modification) is returned in the space to which it points. In this way, with a NULL *set* and **SIG_NOP** *operation* the user can determine the current signal mask.

Routines described in *sigsetops*(3) are used to create and examine the *set* and *oset* signal masks.

It is not possible to block **SIGKILL** or **SIGSTOP**; this restriction is silently imposed by the system.

POSIX specifies (contrary to BSD and System V) that a process *may* block **SIGCONT**. However, a) **SIGCONT** *always* restarts the receiving process (unless it is waiting for an event such as I/O), and b) if the receiving process has installed a handler for **SIGCONT** and blocked the signal, the process will NOT enter its handler until it unblocks **SIGCONT**. (The signal will remain pending.)

sigprocmask will fail if:

[EFAULT]	*set* or *oset* point to memory that is not a part of the process's valid address space.
[EINVAL]	*Operation* is not a valid set-operation (as described above: **SIG_NOP**, **SIG_BLOCK**, **SIG_UNBLOCK**, or **SIG_SETMASK**).

SEE ALSO

kill(2), sigaction(2), sigpending(2), sigsuspend(2), sigsetops(3).

DIAGNOSTICS

A 0 value indicates that the call succeeded. A –1 return value indicates that an error occurred and *errno* is set to indicate the reason.

WARNING

The POSIX and System V signal facilities have different semantics. Using both facilities in the same program is **strongly discouraged** and will result in unpredictable behavior.

NAME

sigset, sighold, sigrelse, sigignore, sigpause – signal management (System V)

C SYNOPSIS

#include <signal.h>

void (*sigset (int sig, void (*func)(int, ...)))(int, ...);

int sighold (int sig);

int sigrelse (int sig);

int sigignore (int sig);

int sigpause (int sig);

DESCRIPTION

These functions provide signal management for application processes. *sigset* specifies the system signal action to be taken upon receipt of signal *sig*. This action is either calling a process signal-catching handler *func* or performing a system-defined action.

sighold and *sigrelse* are used to establish critical regions of code. *sighold* is analogous to raising the priority level and deferring or holding a signal until the priority is lowered by *sigrelse*. *sigrelse* restores the system signal action to that specified previously by *sigset*.

sigignore sets the action for signal *sig* to **SIG_IGN** (see below).

sigpause suspends the calling process until it receives a signal, the same as *pause*(2). However, if the signal *sig* had been received and held, it is released and the system signal action taken. This system call is useful for testing variables that are changed on the occurrence of a signal. The correct usage is to use *sighold* to block the signal first, then test the variables. If they have not changed, then call *sigpause* to wait for the signal.

Sig can be assigned any one of the following values except **SIGKILL** and **SIGSTOP**:

SIGHUP	01	hangup
SIGINT	02	interrupt
SIGQUIT	03[1]	quit
SIGILL	04[1]	illegal instruction (not reset when caught)
SIGTRAP	05[1][5]	trace trap (not reset when caught)
SIGABRT	06[1]	abort
SIGEMT	07[1][4]	EMT instruction
SIGFPE	08[1]	floating point exception
SIGKILL	09	kill (cannot be caught or ignored)

SIGBUS	10[1]	bus error
SIGSEGV	11[1]	segmentation violation
SIGSYS	12[1]	bad argument to system call
SIGPIPE	13	write on a pipe with no one to read it
SIGALRM	14	alarm clock
SIGTERM	15	software termination signal
SIGUSR1	16	user-defined signal 1
SIGUSR2	17	user-defined signal 2
SIGCLD	18[2]	death of a child
SIGPWR	19[2]	power fail (not reset when caught)
SIGSTOP	20[6]	stop (cannot be caught or ignored)
SIGTSTP	21[6]	stop signal generated from keyboard
SIGPOLL	22[3]	selectable event pending
SIGIO	23[2]	input/output possible
SIGURG	24[2]	urgent condition on IO channel
SIGWINCH	25[2]	window size changes
SIGVTALRM	26	virtual time alarm
SIGPROF	27	profiling alarm
SIGCONT	28[6]	continue after stop (cannot be ignored)
SIGTTIN	29[6]	background read from control terminal
SIGTTOU	30[6]	background write to control terminal
SIGXCPU	31	cpu time limit exceeded [see *setrlimit*(2)]
SIGXFSZ	32	file size limit exceeded [see *setrlimit*(2)]

Func is assigned one of four values: **SIG_DFL**, **SIG_IGN**, or **SIG_HOLD**, which are macros (defined in <*sys/signal.h*>) that expand to constant expressions, or a *function address*.

The actions prescribed by its value are as follows:

SIG_DFL – terminate process upon receipt of a signal

Upon receipt of the signal *sig*, the receiving process is to be terminated with all of the consequences outlined in *exit*(2). See SIGNAL NOTES [1] below.

SIG_IGN – ignore signal

The signal *sig* is to be ignored.

Note: the signals **SIGKILL, SIGSTOP** and **SIGCONT** cannot be ignored.

SIG_HOLD – hold signal

The signal *sig* is to be held upon receipt. Any pending signal of this type remains held. Only one signal of each type is held.

function address – catch signal

Upon receipt of the signal *sig*, the receiving process is to execute the signal-catching function whose address is specified via this parameter. The function will be invoked as follows:

handler (int sig, int code, struct sigcontext *sc);

Where *handler* is the specified function-name. *code* is valid only in the following cases:

Condition	Signal	Code
User breakpoint	SIGTRAP	BRK_USERBP
User breakpoint	SIGTRAP	BRK_SSTEPBP
Integer overflow	SIGTRAP	BRK_OVERFLOW
Divide by zero	SIGTRAP	BRK_DIVZERO
Multiply overflow	SIGTRAP	BRK_MULOVF
Invalid virtual address	SIGSEGV	EFAULT
Read-only address	SIGSEGV	EACCESS
Read beyond mapped object	SIGSEGV	ENXIO

The third argument *sc* is a pointer to a *struct sigcontext* (defined in *<sys/signal.h>*) that contains the processor context at the time of the signal.

Before the handler is invoked the signal action will be changed to **SIG_HOLD**.

The signal-catching function remains installed after it is invoked. During normal return from the signal-catching handler, the system signal action is restored to *func* and any held signal of this type released. If a non-local goto (*longjmp*) is taken, then *sigrelse* must be called to restore the system signal action and release any held signal of this type.

Upon return from the signal-catching function, the receiving process will resume execution at the point it was interrupted. See WARNINGS below.

When a signal that is to be caught occurs during a *read*(2), a *write*(2), an *open*(2), or an *ioctl*(2) system call on a slow device (like a terminal; but not a file), during a *pause*(2) system call, or during a *wait*(2) system call that does not return immediately due to the existence of a previously stopped or zombie process, the signal catching function will be executed and then the interrupted system call may return a –1 to the calling process with *errno* set to **EINTR**.

Note: The signals **SIGKILL** and **SIGSTOP** cannot be caught.

SIGNAL NOTES

[1] If **SIG_DFL** is assigned for these signals, in addition to the process being terminated, a "core image" will be constructed in the current working directory of the process, if the following conditions are met:

The effective user ID and the real user ID of the receiving process are equal.

An ordinary file named **core** exists and is writable or can be created. If the file must be created, it will have the following properties:

a mode of 0666 modified by the file creation mask [see *umask*(2)]

a file owner ID that is the same as the effective user ID of the receiving process.

a file group ID that is the same as the effective group ID of the receiving process

NOTE: The core file may be truncated if the resultant file size would exceed either *ulimit* [see *ulimit*(2)] or the process's maximum core file size [see *setrlimit*(2)].

[2] For the signals **SIGCLD, SIGWINCH, SIGPWR, SIGURG,** and **SIGIO,** the handler parameter is assigned one of three values: **SIG_DFL, SIG_IGN,** or a *function address*. The actions prescribed by these values are:

SIG_DFL - ignore signal
The signal is to be ignored.

SIG_IGN - ignore signal
The signal is to be ignored. Also, if *sig* is **SIGCLD**, the calling process's child processes will not create zombie processes when they terminate [see *exit*(2)].

function address - catch signal
If the signal is **SIGPWR, SIGWINCH, SIGURG,** or **SIGIO,** the action to be taken is the same as that described above for a handler parameter equal to *function address*. The same is true if the signal is **SIGCLD** with one exception: while the process is executing the signal-catching function, all terminating child processes will be queued. The *wait* system call removes the first entry of the queue. To ensure that no **SIGCLD**'s are missed while executing in a **SIGCLD** handler, it is necessary to call *sigset* to re-attach

the handler before exiting from it, and at that time--if the queue is not empty--**SIGCLD** is re-raised before *sigset* returns. See *wait*(2). If the signal handler is simply exited from, then **SIGCLD** will NOT be re-raised automatically.

In addition, **SIGCLD** affects the *wait* and *exit* system calls as follows:

wait If the handler parameter of **SIGCLD** is set to **SIG_IGN** and a *wait* is executed, the *wait* will block until all of the calling process's child processes terminate; it will then return a value of −1 with *errno* set to **ECHILD.**

exit If in the exiting process's parent process the handler parameter of **SIGCLD** is set to **SIG_IGN**, the exiting process will not create a zombie process.

When processing a pipeline, the shell makes the last process in the pipeline the parent of the proceeding processes. A process that may be piped into in this manner (and thus become the parent of other processes) should take care not to set **SIGCLD** to be caught.

[3] **SIGPOLL** is issued when a file descriptor corresponding to a STREAMS [see *intro*(2)] file has a "selectable" event pending. A process must specifically request that this signal be sent using the **I_SETSIG** *ioctl* call. Otherwise, the process will never receive **SIGPOLL.**

[4] **SIGEMT** is never generated on an **IRIS-4D** system.

[5] **SIGTRAP** is generated for breakpoint instructions, overflows, divide by zeros, range errors, and multiply overflows. The second argument *code* gives specific details of the cause of the signal. Possible values are described in *<sys/signal.h>*.

[6] The signals **SIGSTOP, SIGTSTP, SIGTTIN, SIGTTOU** and **SIGCONT** are used by command interpreters like the C shell [see *csh*(1)] to provide job control. The first four signals listed will cause the receiving process to be stopped, unless the signal is caught or ignored. **SIGCONT** causes a stopped process to be resumed. **SIGTSTP** is sent from the terminal driver in response to the SWTCH character being entered from the keyboard [see *termio*(7)]. **SIGTTIN** is sent from the terminal driver when a background process attempts to read from its controlling terminal. If **SIGTTIN** is ignored by the process, then the read will return **EIO. SIGTTOU** is sent from the terminal driver when a background process attempts to write to its controlling terminal when the terminal is in **TOSTOP** mode. If **SIGTTOU** is ignored by the process, then the write will succeed regardless of the state of the

controlling terminal.

NOTES

SIGKILL will immediately terminate a process, regardless of its state. Processes which are stopped via job control (<ctrl>z) will not act upon any delivered signals other than SIGKILL until the job is restarted. Processes which are blocked via a *blockproc* system call will unblock if they receive a signal which is fatal (i.e. a non-job-control signal which the are NOT catching), but will still be stopped if the job of which they are a part is stopped. Only upon restart will they die. Any non-fatal signals received by a blocked process will NOT cause the process to be unblocked (an *unblockproc* or *unblockprocall* system call is necessary).

After a *fork*(2) the child inherits all handlers and signal masks, but not the set of pending signals.

The *exec*(2) routines reset all caught signals to the default action; ignored signals remain ignored, the blocked signal mask is unchanged and pending signals remain pending.

sigset will fail if one or more of the following are true:

[EINVAL] *Sig* is an illegal signal number (including SIGKILL and SIGSTOP) or the default handling of *sig* cannot be changed.

[EINVAL] The requested action is illegal (e.g. ignoring SIGCONT, which is ignored by default).

[EINTR] A signal was caught during the system call *sigpause*.

DIAGNOSTICS

Upon successful completion, *sigset* returns the previous value of the system signal action for the specified signal *sig*. Otherwise, a value of SIG_ERR is returned and *errno* is set to indicate the error. SIG_ERR is defined in *<sys/signal.h>*.

For the other functions, upon successful completion, a value of 0 is returned. Otherwise, a value of −1 is returned and *errno* is set to indicate the error.

SEE ALSO

csh(1), kill(2), pause(2), setrlimit(2), signal(2), ulimit(2), wait(2), sigaction(2), setjmp(3C), sigvec(3B), blockproc(2).

WARNINGS

Signals raised by the instruction stream, SIGILL, SIGEMT, SIGBUS, SIGSEGV will cause infinite loops if their handler returns, or the action is set to SIG_IGN.

WARNING

The POSIX signal routines (*sigaction*(2), *sigpending*(2), *sigprocmask*(2), *sigsuspend*(2), *sigsetjmp*(3)), and the 4.3BSD signal routines (*sigvec*(3B), *signal*(3B), *sigblock*(3B), *sigpause*(3B), *sigsetmask*(3B)) must NEVER be used with *signal*(2) or *sigset*(2).

NAME

sigsuspend – atomically release blocked signals and wait for interrupt (POSIX)

SYNOPSIS

int sigsuspend(sigset_t *maskptr);

DESCRIPTION

sigsuspend replaces the process's set of masked signals with the set pointed to by *maskptr* and then waits for a signal to arrive; upon return the original set of masked signals is restored after executing the handler(s) (if any) installed for the awakening signal(s). The specified signal mask is usually 0 to indicate that no signals are to be blocked during the wait. *sigsuspend* always terminates by being interrupted, returning −1 with *errno* set to EINTR.

In normal usage, a signal is blocked via *sigprocmask*(2) to begin a critical section, variables modified on the occurrence of the signal are examined to determine that there is no work to be done, and the process pauses by calling *sigsuspend* with the mask returned by *sigprocmask*.

Routines described in *sigsetops*(3) are used to create and manipulate the input-parameter signal masks submitted to *sigaction*(2), *sigprocmask*(2), and *sigsuspend*(2), and returned by *sigpending*(2). These masks are of type *sigset_t*.

sigsuspend will fail if:

[EFAULT] *maskptr* points to memory that is not a part of process's valid address space.

NOTES

POSIX specifies (contrary to BSD and System V) that a process *may* block SIGCONT. However, a) SIGCONT *always* restarts the receiving process (unless it is waiting for an event such as I/O), and b) if the receiving process has installed a handler for SIGCONT and blocked the signal, the process will NOT enter its handler until it unblocks SIGCONT. (The signal will remain pending.) Therefore, if *sigsuspend* is called with a mask which blocks SIGCONT, receipt of that signal will set the process running, but *not* cause it to enter a handler.

SEE ALSO

sigaction(2), sigpending(2), sigprocmask(2), sigsetops(3).

WARNING

The POSIX and System V signal facilities have different semantics. Using both facilities in the same program is **strongly discouraged** and will result in unpredictable behavior.

NAME

socket – create an endpoint for communication

C SYNOPSIS

#include <sys/types.h>
#include <sys/socket.h>

int socket(int domain, int type, int protocol)

DESCRIPTION

Socket creates an endpoint for communication and returns a descriptor.

The *domain* parameter specifies a communications domain within which communication will take place; this selects the protocol family which should be used. The protocol family generally is the same as the address family for the addresses supplied in later operations on the socket. These families are defined in the include file *<sys/socket.h>*. The currently understood formats are:

PF_INET	(DARPA Internet protocols)
PF_RAW	(Link-level protocols)
PF_UNIX	(4.3BSD UNIX internal protocols)

The following are defined but currently unimplemented:

PF_NS	(Xerox Network Systems protocols), and
PF_IMPLINK	(IMP "host at IMP" link layer).

The socket has the indicated *type*, which specifies the semantics of communication. Currently defined types are:

SOCK_STREAM
SOCK_DGRAM
SOCK_RAW
SOCK_SEQPACKET
SOCK_RDM

A SOCK_STREAM type provides sequenced, reliable, two-way connection based byte streams. An out-of-band data transmission mechanism may be supported. A SOCK_DGRAM socket supports datagrams (connectionless, unreliable messages of a fixed (typically small) maximum length). SOCK_RAW sockets, which are available only to the super-user, provide access to internal network protocols and interfaces. The types SOCK_SEQPACKET and SOCK_RDM are currently unimplemented.

The *protocol* specifies a particular protocol to be used with the socket. Normally only a single protocol exists to support a particular socket type within a given protocol family. However, it is possible that many protocols may exist, in which case a particular protocol must be specified in this manner.

The protocol number to use is particular to the "communication domain" in which communication is to take place; see *getprotoent*(3N).

Sockets of type SOCK_STREAM are full-duplex byte streams, similar to pipes. A stream socket must be in a *connected* state before any data may be sent or received on it. A connection to another socket is created with a *connect*(2) call. Once connected, data may be transferred using *read*(2) and *write*(2) calls or some variant of the *send*(2) and *recv*(2) calls. Note that for the *read* and *recv*-style calls, the number of bytes actually read may be less than the number requested. When a session has been completed a *close*(2) may be performed. Out-of-band data may also be transmitted as described in *send*(2) and received as described in *recv*(2).

The communications protocols used to implement a SOCK_STREAM insure that data is not lost or duplicated. If a piece of data for which the peer protocol has buffer space cannot be successfully transmitted within a reasonable length of time, then the connection is considered broken and calls will indicate an error with −1 returns and with ETIMEDOUT as the specific code in the global variable errno. The protocols optionally keep sockets "warm" by forcing transmissions roughly every minute in the absence of other activity. An error is then indicated if no response can be elicited on an otherwise idle connection for a extended period (e.g. 5 minutes). A SIGPIPE signal is raised if a process sends on a broken stream; this causes naive processes, which do not handle the signal, to exit.

SOCK_DGRAM and SOCK_RAW sockets allow sending of datagrams to correspondents named in *send*(2) calls. Datagrams are generally received with *recvfrom*(2), which returns the next datagram with its return address.

An *fcntl*(2) call can be used to specify a process group to receive a SIGURG signal when the out-of-band data arrives. The FIONBIO i/o control (see *ioctl*(2)) or the FNDELAY fcntl (see *fcntl*(2)) enable non-blocking I/O and asynchronous notification of I/O events via SIGIO.

The operation of sockets is controlled by socket level *options*. These options are defined in the file *<sys/socket.h>*. *setsockopt*(2) and *getsockopt*(2) are used to set and get options, respectively.

RETURN VALUE

A −1 is returned if an error occurs, otherwise the return value is a descriptor referencing the socket.

ERRORS

The *socket* call fails if:

[EPROTONOSUPPORT]	The protocol type or the specified protocol is not supported within this domain.
[EMFILE]	The per-process descriptor table is full.
[ENFILE]	The system file table is full.
[EACCESS]	Permission to create a socket of the specified type and/or protocol is denied.
[ENOBUFS]	Insufficient buffer space is available. The socket cannot be created until sufficient resources are freed.

SEE ALSO

accept(2), bind(2), connect(2), fcntl(2), getsockname(2), getsockopt(2), ioctl(2), listen(2), read(2), recv(2), select(2), send(2), socketpair(2), write(2)

Network Programming chapter in the *Network Communications Guide*.

NAME

socketpair – create a pair of connected sockets

SYNOPSIS

#include <sys/types.h>
#include <sys/socket.h>

socketpair(d, type, protocol, sv)
int d, type, protocol;
int sv[2];

DESCRIPTION

The *socketpair* call creates an unnamed pair of connected sockets in the specified domain *d*, of the specified *type*, and using the optionally specified *protocol*. The descriptors used in referencing the new sockets are returned in *sv*[0] and *sv*[1]. The two sockets are indistinguishable.

DIAGNOSTICS

A 0 is returned if the call succeeds, –1 if it fails.

ERRORS

The call succeeds unless:

[EMFILE]	Too many descriptors are in use by this process.
[EAFNOSUPPORT]	The specified address family is not supported on this machine.
[EPROTONOSUPPORT]	The specified protocol is not supported on this machine.
[EOPNOSUPPORT]	The specified protocol does not support creation of socket pairs.
[EFAULT]	The address *sv* does not specify a valid part of the process address space.

SEE ALSO

read(2), write(2), pipe(2)

BUGS

This call is currently implemented only for the UNIX domain.

NAME

sproc – create a new share group process

C SYNOPSIS

#include <sys/types.h>
#include <sys/prctl.h>

int sproc (void *entry (void *), unsigned inh, ...);

Type of optional third argument:
void *arg;

DESCRIPTION

The *sproc* system call is a variant of the standard *fork*(2) call. Like *fork*, *sproc* creates a new process that is a clone of the process that called *sproc*. The difference is that after an *sproc*, the new child process shares the virtual address space of the parent process (assuming that this sharing option is selected, as described below), rather than simply being a copy of the parent. The parent and the child each have their own program counter value and stack pointer, but all the text and data space is visible to both processes. This provides one of the basic mechanisms upon which parallel programs can be built.

A group of processes created by *sproc* calls from a common ancestor is referred to as a *share group* or *shared process group*. A share group is initially formed when a process first executes an *sproc* call. All subsequent *sproc* calls by either the parent or other children in his share group will add another process to the share group. In addition to virtual address space, members of a share group can share other attributes such as file tables, current working directories, effective userids and others described below.

The new child process resulting from *sproc*(2) differs from a normally forked process in the following ways:

The child's stack is set to a virtual address that doesn't overlap the stack of the parent process. There is a maximum stack size different from the maximum allowable amount of virtual space per process. This value may be read and set using *prctl*(2) or *setrlimit*(2).

If the **PR_SADDR** bit is set in *inh* then the new process will share ALL the virtual space of the parent, except the PRDA (see below). During a normal *fork*(2), the writable portions of the process's address space are marked copy-on-write. If either process writes into a given page, then a copy is made of the page and given to the process. Thus writes by one process will not be visible to the other forks. With the **PR_SADDR** option of *sproc(2)*, however, all the processes have read/write privileges to the entire virtual space.

The new process can reference the parent's stack.

The new process has its own *process data area* (PRDA) which contains, among other things, the *process id*. Part of the PRDA is used by the system, part by system libraries, and part is available to the application program [see **<sys/prctl.h>**]. The PRDA is at a fixed virtual address in each process which is given by the constant **PRDA** defined in **prctl.h**.

The machine state (general/floating point registers) is not duplicated with the exception of the floating point control register. This means that if a process has enabled floating point traps, these will be enabled in the child process.

The new process will be invoked as follows:

entry(arg)

In addition to the attributes inherited during the *sproc* call itself, the *inh* flag to *sproc* can request that the new process have future changes in any member of the share group be applied to itself. A process can only request that a child process share attributes that it itself is sharing. The creator of a share group is effectively sharing everything. These persisting attributes are selectable via the *inh* flag:

PR_SADDR All virtual space attributes (shared memory, mapped files, data space) are shared. If one process in a share group attaches to a shared memory segment, all processes in the group can access that segment.

PR_SFDS The open file table is kept synchronized. If one member of the share group opens a file, the open file descriptor will appear in the file tables of all members of the share group. Note that there is only one file pointer for each file descriptor shared within a shared process group.

PR_SDIR The current and root directories are kept synchronized. If one member of the group issues a *chdir*(2) or *chroot*(2) call, the current working directory or root directory will be changed for all members of the share group.

PR_SUMASK The file creation mask, *umask* is kept synchronized.

PR_SULIMIT The limit on maximum file size is kept synchronized.

PR_SID The real and effective user and group ids are kept synchronized.

To take advantage of sharing all possible attributes, the constant **PR_SALL** may be used.

In addition to specifying shared attributes, the *inh* flag can be used to pass flags that govern certain operations within the *sproc* call itself. Currently one flag is supported, **PR_BLOCK**, which causes the calling process to be blocked [see *blockproc*(2)] before returning from a successful call. This can be used to allow the child process access to the parent's stack without the possibility of collision.

No scheduling synchronization is implied between shared processes: they are free to run on any processor in any sequence. Any required synchronization must be provided by the application using locks and semaphores [see *usinit*(3P)] or other mechanisms.

If one member of a share group exits or otherwise dies, its stack is removed from the virtual space of the share group. In addition, if the **PR_SETEXITSIG** option [see *prctl*(2)] has been enabled then all remaining members of the share group will be signaled.

There are two versions of *sproc*, one in **libc.a** and one in **libmpc.a**. Users linking with the semaphored version of libc, **libmpc.a**, by using the **-lmpc** flag to the compiler, will have standard routines such as *printf* and *malloc* function properly even though two or more shared processes access them simultaneously. To accomplish this, a special arena is set up [see *usinit*(3P)] to hold the locks and semaphores required. Each process in the share group needs access to this arena and requires a single file lock [see *fcntl*(2)]. This may require more file locks to be configured into the system than the default system configuration provides.

sproc will fail and no new process will be created if one or more of the following are true:

[ENOMEM] If there is not enough virtual space to allocate a new stack. The default stack size is settable via *prctl*(2), or *setrlimit*(2).

[EAGAIN] The system-imposed limit on the total number of processes under execution, *{NPROC}* [see *intro*(2)], would be exceeded.

[EAGAIN] The system-imposed limit on the total number of processes under execution by a single user *{CHILD_MAX}* [see *intro*(2)], would be exceeded.

[EAGAIN] Amount of system memory required is temporarily unavailable.

When linked with **libmpc.a**, in addition to the above errors *sproc* will fail and no new process will be created if one or more of the following are true:

[ENOSPC] If the size of the share group exceeds the number of users specified via *usconfig*(3P) (8 by default). Any changes via *usconfig*(3P) must be done BEFORE the first *sproc* is performed.

[ENOLCK] There are not enough file locks in the system.

New share group member pid # could not join I/O arena. error:<..> if the new share group member could not properly join the semaphored libc arena. The new process exits with a −1.

See also the possible errors from *usinit(3P)*.

NOTES

This manual entry has described ways in which processes created by *sproc* differ from those created by *fork*. Attributes and behavior not mentioned as different should be assumed to work the same way for *sproc* processes as for processes created by *fork*. Here are some respects in which the two types of processes are the same:

The parent and child after an *sproc* each have a unique process id (*pid*), but are in the same process group.

A signal sent to a specific *pid* in a share group [see *kill*(2)] will be received by only the process to which it was sent. Other members of the share group will not be affected. A signal sent to an entire process group will be received by all the members of the process group, regardless of share group affiliations [see *killpg*(3B)]. See *prctl*(2) for ways to alter this behavior.

If the child process resulting from an *sproc* dies or calls *exit*(2), the parent process receives the SIGCLD signal [see *sigset*(2), *sigaction*(2), and *sigvec*(3B)].

CAVEATS

Removing virtual space (e.g. unmapping a file) is an expensive operation and effectively forces all processes in the share group to single thread.

Note that the global variable *errno* is shared by all processes in an *sproc* share group in which address space is a shared attribute. This means that if multiple processes in the group make system calls, the value of *errno* is no longer useful, since it may be overwritten at any time by a system call in another process in the share group. In order to allow a process in a share group to determine the value of *errno* reliably, the system call modules in **libmpc.a** store the error return code in a location in the PRDA that is private to each process in the share group, in addition to storing it in the

global variable *errno*. A library routine *oserror*(3C) is provided in both **libc.a** and **libmpc.a** that returns the current value *errno* for the process making the call.

SEE ALSO

blockproc(2), fcntl(2), fork(2), prctl(2), setrlimit(2), oserror(3C), pcreate(3C), usconfig(3P), usinit(3P).

DIAGNOSTICS

Upon successful completion, *sproc* returns the process id of the new process. Otherwise, a value of −1 is returned to the calling process, and *errno* is set to indicate the error.

NAME

stat, lstat, fstat – get file status

C SYNOPSIS

#include <sys/types.h>
#include <sys/stat.h>

int stat (const char *path, struct stat *buf);

int lstat (const char *path, struct stat *buf);

int fstat (int fildes, struct stat *buf);

DESCRIPTION

Path points to a path name naming a file. Read, write or execute permission of the named file is not required, but all directories listed in the path name leading to the file must be searchable. *stat* obtains information about the named file.

lstat is like *stat* except in the case where the named file is a symbolic link, in which case *lstat* returns the information about the link, while *stat* returns the information about the file the link references.

Similarly, *fstat* obtains information about an open file known by the file descriptor *fildes*, obtained from a successful *open*, *creat*, *dup*, *fcntl*, or *pipe* system call.

Buf is a pointer to a *stat* structure into which information is placed concerning the file.

The contents of the structure pointed to by *buf* include the following members:

```
struct stat {
        dev_t   st_dev;     /* ID of device containing */
                            /* a directory entry for this file */
        ino_t   st_ino;     /* Inode number */
        mode_t  st_mode;    /* File mode; see mknod(2) */
        short   st_nlink;   /* Number of links */
        ushort  st_uid;     /* User ID of the file's owner */
        ushort  st_gid;     /* Group ID of the file's group */
        dev_t   st_rdev;    /* ID of device */
                            /* This entry is defined only for */
                            /* character special or block special files */
        off_t   st_size;    /* File size in bytes */
                            /* or, for fstat on block devices, */
                            /* device size in 512-byte blocks */
        time_t  st_atime;   /* Time of last access */
        time_t  st_mtime;   /* Time of last data modification */
        time_t  st_ctime;   /* Time of last file status change */
```

```
                    /* Times measured in seconds since */
                    /* 00:00:00 GMT, Jan. 1, 1970 */
};
```

st_atime
Time when file data was last accessed. Changed by the following system calls: *creat*(2), *mknod*(2), *pipe*(2), *utime*(2), and *read*(2).

st_mtime
Time when data was last modified. Changed by the following system calls: *creat*(2), *mknod*(2), *pipe*(2), *utime*(2), and *write*(2).

st_ctime
Time when file status was last changed. Changed by the following system calls: *chmod*(2), *chown*(2), *creat*(2), *link*(2), *mknod*(2), *pipe*(2), *unlink*(2), *utime*(2), and *write*(2).

Note: the st_size field is set for block devices only by *fstat* and not by *stat*. It is set only for block device files which are associated with a real disk device.

stat and *lstat* will fail if one or more of the following are true:

[ENOTDIR]	A component of the path prefix is not a directory.
[ENOENT]	The named file does not exist.
[EACCES]	Search permission is denied for a component of the path prefix.
[ENAMETOOLONG]	The length of *path* exceeds *{PATH_MAX}*, or a pathname component is longer than *{NAME_MAX}*.
[ELOOP]	Too many symbolic links were encountered in translating the pathname.
[EFAULT]	*Buf* or *path* points to an invalid address.

fstat will fail if one or more of the following are true:

[EBADF]	*Fildes* is not a valid open file descriptor.
[EFAULT]	*Buf* points to an invalid address.

SEE ALSO

chmod(2), chown(2), creat(2), link(2), mknod(2), time(2), truncate(2), unlink(2), utime(2), utimes(3B).

DIAGNOSTICS

Upon successful completion a value of 0 is returned. Otherwise, a value of −1 is returned and *errno* is set to indicate the error.

NAME

statfs, fstatfs – get file system information

SYNOPSIS

```
#include <sys/types.h>
#include <sys/statfs.h>

int statfs (path, buf, len, fstyp)
char *path;
struct statfs *buf;
int len, fstyp;

int fstatfs (fildes, buf, len, fstyp)
int fildes;
struct statfs *buf;
int len, fstyp;
```

DESCRIPTION

statfs returns a ''generic superblock'' describing a file system. It can be used to acquire information about mounted as well as unmounted file systems, and usage is slightly different in the two cases. In all cases, *buf* is a pointer to a structure (described below) which will be filled by the system call, and *len* is the number of bytes of information which the system should return in the structure. *Len* must be no greater than **sizeof (struct statfs)** and ordinarily it will contain exactly that value; if it holds a smaller value the system will fill the structure with that number of bytes. (This allows future versions of the system to grow the structure without invalidating older binary programs.)

If the file system of interest is currently mounted, *path* should name a file which resides on that file system. In this case the file system type is known to the operating system and the *fstyp* argument must be zero. For an unmounted file system *path* must name the block special file containing it and *fstyp* must contain the (non-zero) file system type. In both cases read, write, or execute permission of the named file is not required, but all directories listed in the path name leading to the file must be searchable.

The *statfs* structure pointed to by *buf* includes the following members:

```
short   f_fstyp;    /* File system type */
long    f_bsize;    /* Block size */
long    f_frsize;   /* Fragment size */
long    f_blocks;   /* Total number of blocks */
long    f_bfree;    /* Count of free blocks */
long    f_files;    /* Total number of file nodes */
long    f_ffree;    /* Count of free file nodes */
char    f_fname[6]; /* Volume name */
char    f_fpack[6]; /* Pack name */
```

fstatfs is similar, except that the file named by *path* in *statfs* is instead identified by an open file descriptor *filedes* obtained from a successful *open*(2), *creat*(2), *dup*(2), *fcntl*(2), or *pipe*(2) system call.

statfs obsoletes *ustat*(2) and should be used in preference to it in new programs.

statfs and *fstatfs* will fail if one or more of the following are true:

[ENOTDIR]	A component of the path prefix is not a directory.
[ENOENT]	The named file does not exist.
[EACCES]	Search permission is denied for a component of the path prefix.
[EFAULT]	*Buf* or *path* points to an invalid address.
[EBADF]	*Fildes* is not a valid open file descriptor.
[EINVAL]	*Fstyp* is an invalid file system type; *path* is not a block special file and *fstyp* is nonzero; *len* is negative or is greater than **sizeof (struct statfs)**.

DIAGNOSTICS

Upon successful completion a value of 0 is returned. Otherwise, a value of −1 is returned and *errno* is set to indicate the error.

SEE ALSO

chmod(2), chown(2), creat(2), link(2), mknod(2), pipe(2), read(2), time(2), unlink(2), utime(2), write(2), fs(4).

NAME

stime – set time

C SYNOPSIS

int stime (tp)
long *tp;

DESCRIPTION

stime sets the system's idea of the time and date. *Tp* points to the value of time as measured in seconds from 00:00:00 GMT January 1, 1970.

[EPERM] *stime* will fail if the effective user ID of the calling process is not super-user.

SEE ALSO

time(2), gettimeofday(3B), ctime(3C).

DIAGNOSTICS

Upon successful completion, a value of 0 is returned. Otherwise, a value of −1 is returned and *errno* is set to indicate the error.

NAME

symlink – make symbolic link to a file

C SYNOPSIS

symlink (name1, name2)
char *name1, *name2;

DESCRIPTION

A symbolic link *name2* is created to *name1* (*name2* is the name of the file created, *name1* is the string used in creating the symbolic link). Either name may be an arbitrary path name; the files need not be on the same file system.

RETURN VALUE

Upon successful completion, a zero value is returned. If an error occurs, the error code is stored in *errno* and a –1 value is returned.

ERRORS

The symbolic link is made unless on or more of the following are true:

[ENOTDIR]	A component of the *name2* prefix is not a directory.
[ENOENT]	A component of the *name2* prefix does not exist.
[EEXIST]	*Name2* already exists.
[EACCES]	A component of the *name2* path prefix denies search permission.
[EROFS]	The file *name2* would reside on a read-only file system.
[EFAULT]	*Name1* or *name2* points outside the process's allocated address space.
[ELOOP]	Too may symbolic links were encountered in translating the pathname.

SEE ALSO

link(2), ln(1), readlink(2), unlink(2)

NAME

sync – update super block

C SYNOPSIS

void sync ( )

DESCRIPTION

sync causes all information in memory that should be on disk to be written out. This includes modified super blocks, modified i-nodes, and delayed block I/O.

It should be used by programs which examine a file system, for example *fsck*, *df*, etc. It is mandatory before a re-boot.

The writing, although scheduled, is not necessarily complete upon return from *sync*.

NAME

sysconf - get configurable system variables (POSIX)

C SYNOPSIS

#include <unistd.h>

long sysconf (int name);

DESCRIPTION

The *sysconf* function provides a method for an application to determine the current value of a configurable system limit or option (*variable*).

The *name* argument represents the system variable to be queried. The following table lists the variables to be queried (compile-time values of which appear in **<limits.h>**, **<unistd.h>**, or **<time.h>**) on the left, and the *names* used to retrieve them (defined in **<unistd.h>**) on the right:

{ARG_MAX}	**_SC_ARG_MAX**
{CHILD_MAX}	**_SC_CHILD_MAX**
{CLK_TCK}	**_SC_CLK_TCK**
{NGROUPS_MAX}	**_SC_NGROUPS_MAX**
{OPEN_MAX}	**_SC_OPEN_MAX**
{_POSIX_JOB_CONTROL}	**_SC_JOB_CONTROL**
{_POSIX_SAVED_IDS}	**_SC_SAVED_IDS**
{_POSIX_VERSION}	**_SC_VERSION**

sysconf will fail if the following is true:

[EINVAL] The value of the *name* argument is invalid.

SEE ALSO

pathconf(2), limits(4).

DIAGNOSTICS

Upon successful completion, the *sysconf* function returns the current variable value on the system. The value returned will never be more restrictive than the corresponding value described to the application when it was compiled with the implementation's <limits.h> or <unistd.h>, and the value will not change during the lifetime of the calling process.

If unsuccessful, *sysconf* returns –1 and *errno* is set to the appropriate error.

NAME

sysfs – get file system type information

SYNOPSIS

```
#include <sys/fstyp.h>
#include <sys/fsid.h>

int sysfs (opcode, fsname)
int opcode;
char *fsname;

int sysfs (opcode, fs_index, buf)
int opcode;
int fs_index;
char *buf;

int sysfs (opcode)
int opcode;
```

DESCRIPTION

sysfs returns information about the file system types configured in the system. The number of arguments accepted by *sysfs* varies and depends on the *opcode*. The currently recognized *opcodes* and their functions are described below:

GETFSIND translates *fsname*, a null-terminated file-system identifier, into a file-system type index. Possible values of *fsname* can be found in the include file *<sys/fsid.h>*. Note that not all of the names there are currently used: only "com", "nfs", "socket", "efs" and "dbg" are in use. Any other name will result in a return of EINVAL. Note also that "efs" will not be present on diskless systems.

GETFSTYP translates *fs_index*, a file-system type index, into a null-terminated file-system identifier and writes it into the buffer pointed to by *buf*; this buffer must be at least of size **FSTYPSZ** as defined in *<sys/fstyp.h>*.

GETNFSTYP returns the total number of file system types configured in the system.

sysfs will fail if one or more of the following are true:

[EINVAL] *Fsname* points to an invalid file-system identifier; *fs_index* is zero, or invalid; *opcode* is invalid.

[EFAULT] *Buf* or *fsname* point to an invalid user address.

DIAGNOSTICS

Upon successful completion, *sysfs* returns the file-system type index if the *opcode* is **GETFSIND**, a value of 0 if the *opcode* is **GETFSTYP**, or the number of file system types configured if the *opcode* is **GETNFSTYP**. Otherwise, a value of -1 is returned and *errno* is set to indicate the error.

NAME

sysmips – MIPS Computer Systems Inc. system call

SYNOPSIS

#include <sys/sysmips.h>

int sysmips (cmd, arg1, arg2, arg3)
int cmd, arg1, arg2, arg3;

DESCRIPTION

sysmips is the interface to various machine specific functions. The *cmd* argument determines the function performed. The number of arguments expected is dependent on the function.

SETNAME

This function renames the system, which is sometimes referred to as the node name or host name. A single argument of type char * is expected. This points to a string that has a length less or equal to MAXHOSTNAMELEN characters (defined in param.h).

STIME

This function sets the system time and date. The single argument is of type long and contains the time as measured in seconds from 00:00:00 GMT January 1, 1970. It does not affect the hardware battery backed up time-of-day clock. Note that this command is only available to the super-user.

FLUSH_CACHE

This function flushes both the instruction and data caches. On multiprocessor systems, all caches on all processors are flushed. For finer control, use *cacheflush*(2). No arguments are expected.

SMIPSSWPI

Add to, delete from, or determine the currently active swap areas. The address of an appropriately primed swap buffer is passed as the only argument. (This buffer is displayed below and defined in the **sys/swap.h** header file; refer to this file for details of loading the buffer.)

The format of the swap buffer is:

```
struct swapinf {
        char *si_cmd;           /* command: list, add, delete */
        char *si_buf;           /* swap file path pointer */
        int  si_swplo;          /* start block */
        int  si_nblks;          /* swap size */
}
```

Note that the add and delete options of the command may only be exercised by the super-user.

MIPS_FIXADE
This functions tells the operating system to transparently recover from unaligned address exceptions for the current process. For example, a halfword data access on an odd byte boundary would ordinarily generate a **SIGBUS** signal to the user. After calling this function, the operating system will attempt to retry the access using byte-wise instructions.

MIPS_FPSIGINTR
This function permits the caller to decide what happens when a floating point operation requires operating system intervention. If *arg1* is 0, then these floating point operations are silently executed by the operating system. If *arg1* is a 1, then a **SIGFPE** signal is generated before the operation is handled. The process's **SIGFPE** signal handler may then determine exactly what caused the floating point hardware to require software intervention. The operating system, before sending the signal, will change the value from a 1 to a 2 which means the next time an operation requires operating system assistance, the **SIGFPE** will not be generated, rather the value will be set back to a 1, and the operation will be performed. All other values for *arg1* are ignored. This action is cleared on *exec*(2) and inherited on *fork*(2) and *sproc(2)*. A more complete exception handling package may be found in *handle_sigfpes*(3C).

When *cmd* is invalid, *errno* is set to EINVAL on return.

In addition, the *cmd* SETNAME may also return:

[EFAULT] The argument points to an invalid address.

The *cmd* SMIPSSWPI may also return:

[EFAULT] *Swapbuf* points to an invalid address.

[EFAULT] *Swapbuf.si_buf* points to an invalid address.

[ENOTBLK] Swap area specified is not a block special device.

[EEXIST] Swap area specified has already been added.

[ENOSPC] Too many swap areas in use (if adding).

[ENOMEM] Tried to delete last remaining swap area.

[ENOMEM] No place to put swapped pages when deleting a swap area.

[EINVAL] Bad arguments.

SEE ALSO

cachectl(2), cacheflush(2), sethostname(2), signal(2), stime(2), handle_sigfpes(3C), swap(1M).

DIAGNOSTICS

Upon successful completion, the value returned is zero. Otherwise, a value of -1 is returned and *errno* is set to indicate the error.

BUGS

The command code **MIPS_FPU** is defined in *sys/sysmips.h* but is not implemented.

NAME

sysmp – multiprocessing control

C SYNOPSIS

#include <sys/types.h>
#include <sys/sysmp.h>

int sysmp (int cmd, ...);

Possible types of optional second argument:
int arg1;
struct pda_stat *arg1;

Possible types of optional third argument:
void *arg2;

Possible types of optional fourth argument:
int arg3;

Possible types of optional fifth argument:
int arg4;

DESCRIPTION

sysmp provides control/information for miscellaneous system services. This system call is usually used by system programs and is not intended for general use. The arguments *arg1*, *arg2*, *arg3*, *arg4* are provided for command-dependent use.

As specified by *cmd*, the following commands are available:

MP_PGSIZE The page size of the system is returned [see *getpagesize*(2)].

MP_SCHED Interface for the *schedctl*(2) system call.

MP_NPROCS Returns the number of processors physically configured.

MP_NAPROCS Returns the number of processors that are available to schedule unrestricted processes.

MP_STAT The processor ids and status flag bits of the physically configured processors are copied into an array of *pda_stat* structures to which *arg1* points. The array must be large enough to hold as many *pda_stat* structures as the number of processors returned by the **MP_NPROCS** *sysmp* command. The *pda_stat* structure and the various status bits are defined in <*sys/pda.h*>.

MP_EMPOWER Empowers processor numbered *arg1* to run any unrestricted processes. This is the default system configuration for all processors. This command requires superuser authority.

MP_RESTRICT Restricts processor numbered *arg1* from running any processes except those assigned to it by a **MP_MUSTRUN** command, a *runon(1)* command or because of hardware necessity. This command requires superuser authority.

MP_CLOCK Moves the operating system software clock handling to the processor numbered *arg1*. This command requires superuser authority.

MP_MUSTRUN Assigns the calling process to run only on the processor numbered *arg1*, except as required for communications with hardware devices.

MP_RUNANYWHERE
Frees the calling process to run on whatever processor the system deems suitable.

MP_KERNADDR Returns the address of various kernel data structures. The structure returned is selected by *arg1*. The list of available structures is detailed in *<sys/sysmp.h>*. This option is used by many system programs to avoid having to look in **/unix** for the location of the data structures.

MP_SASZ Returns the size of various system accounting structures. As above, the structure returned is governed by *arg1*.

MP_SAGET1 Returns the contents of various system accounting structures. The information is only for the processor specified by *arg4*. As above, the structure returned is governed by *arg1*. *arg2* points to a buffer in the address space of the calling process and *arg3* specifies the maximum number to bytes to transfer.

MP_SAGET Returns the contents of various system accounting structures. The information is summed across all processors before it is returned. As above, the structure returned is governed by *arg1*. *arg2* points to a buffer in the address space of the calling process and *arg3* specifies the maximum number to bytes to transfer.

Possible errors from *sysmp* are:

[EPERM]	The effective user ID is not superuser. Many of the commands require superuser privilege.
[EINVAL]	The processor named by a MP_EMPOWER, MP_RESTRICT, MP_CLOCK or MP_SAGET1 command does not exist.
[EINVAL]	The *cmd* argument is invalid.
[EINVAL]	The *arg1* argument to a MP_KERNADDR command is invalid.
[EBUSY]	An attempt was made to restrict the only unrestricted processor or to restrict the master processor.
[EFAULT]	An invalid buffer address has been supplied by the calling process.

SEE ALSO

mpdmin(1), runon(1), getpagesize(2), schedctl(2).

DIAGNOSTICS

Upon successful completion, the *cmd* dependent data is returned. Otherwise, a value of −1 is returned and *errno* is set to indicate the error.

NAME

syssgi – Silicon Graphics Inc. system call

SYNOPSIS

#include <sys/syssgi.h>

int syssgi (int request, ...);

DESCRIPTION

syssgi is a system interface specific to Silicon Graphics IRIS-4D workstations. The value of the *request* parameter determines the meaning of the remaining arguments. The following requests are currently supported:

SGI_SYSID Returns an identifier for the given system. This identifier is guaranteed to be unique within the Silicon Graphics workstation product family. The argument *arg1* for this *request* should be a pointer to a buffer of MAXSYSIDSIZE characters.

SGI_RDNAME

Returns the process name for the process id specified in *arg1*. The arguments *arg2* and *arg3* give the address and length, respectively, of the buffer which will receive the name string. This name corresponds to the name in the COMMAND column of *ps*(1) for the given process. The returned string will be null-terminated unless the caller's buffer is too small, in which case the string is simply truncated at the size of the buffer. The return value gives the number of bytes copied to the buffer, which will be the minimum of the size of the buffer and the size of the field in the user structure that contains the process name. Note that this means that the returned length will typically be greater than the actual length of the name string (in the sense of *strlen*(3)).

SGI_RDUBLK

Returns the user area for a given process id. The user structure is defined in **<sys/user.h>**. The argument *arg1* should be the process id, argument *arg2* should be a pointer to the buffer to store the user area, and argument *arg3* should be the size of the buffer, not to exceed the page size. The call will return this size upon success, truncated to the system page size if necessary.

SGI_TUNE Either read or write the tune structure, defined in **<sys/tuneable.h>**. This request allows the super-user to redefine virtual memory variables to more accurately reflect system use. The argument *arg1* should be either TUNE_RD

to read the tune structure, or **TUNE_WR** to write it. The argument *arg2* should be a buffer containing the tune structure.

SGI_IDBG Used internally for kernel debugging.

SGI_INVENT Returns information about the hardware inventory of the system. If *arg1* is **SGI_INV_SIZEOF** then the size of an individual inventory item is returned. If *arg1* is **SGI_INV_READ** then *arg3* bytes worth of inventory records are read into the buffer specified by *arg2*.

SGI_SIGACTION

SGI_SIGPENDING

SGI_SIGPROCMASK

SGI_SIGSUSPEND

Internal interfaces for POSIX/BSD signal handling [see *sigaction*(3)].

SGI_SETTIMETRIM

changes the value of timetrim from the initial value configured in **/usr/sysgen/master.d/kernel** [see *lboot*(1M) and *adjtime*(2)].

SGI_GETTIMETRIM

obtains the current value of timetrim.

The following error codes may be returned by *syssgi*:

[EFAULT] A buffer is referenced which is not in a valid part of the calling program's address space.

[ENODEV] Could not determine system ID for **SGI_SYSID**.

[ESRCH] Could not find given process for **SGI_RDUBLK** or **SGI_RDNAME**.

[EPERM] The effective user ID is not super-user. **SGI_TUNE** and **SGI_IDBG** require super-user privilege.

[EINVAL] For **SGI_TUNE**, the first argument was not valid, or the tune structure contained invalid values. For **SGI_INVENT** *arg1* was neither **SGI_INV_READ** nor **SGI_INV_SIZEOF.**

SEE ALSO

ftimer(1), hinv(1), mpadmin(1), adjtime(2), setitimer(2), sigaction(2).

DIAGNOSTICS

Upon successful completion, a command dependent value (default of zero) is returned. Otherwise, a value of −1 is returned and *errno* is set to indicate the error.

NAME

texturebind – SGI graphics system call

DESCRIPTION

This system call is used for underlying operating system support of graphics functions. It is not intended for direct use by user programs.

NAME

time – get time

SYNOPSIS

#include <time.h>

time_t time (time_t *tloc);

DESCRIPTION

time returns the value of time in seconds since 00:00:00 GMT, January 1, 1970.

If *tloc* is non-zero, the return value is also stored in the location to which *tloc* points.

SEE ALSO

stime(2), gettimeofday(3B).

WARNING

time fails and its actions are undefined if *tloc* points to an illegal address.

DIAGNOSTICS

Upon successful completion, *time* returns the value of time. Otherwise, a value of –1 is returned and *errno* is set to indicate the error.

NAME

times – get process and child process times

SYNOPSIS

#include <sys/types.h>
#include <sys/times.h>
#include <sys/param.h>

clock_t times (struct tms *buffer);

DESCRIPTION

times fills the structure pointed to by *buffer* with time-accounting information. The following are the contents of this structure:

```
struct   tms {
             time_t   tms_utime;
             time_t   tms_stime;
             time_t   tms_cutime;
             time_t   tms_cstime;
};
```

This information comes from the calling process and each of its terminated child processes for which it has executed a *wait*. All times are reported in clock ticks per second. Clock ticks are a system-dependent parameter. The specific value for an implementation is defined by the variable HZ, found in the include file *param.h* in */usr/include/sys*.

Tms_utime is the CPU time used while executing instructions in the user space of the calling process.

Tms_stime is the CPU time used by the system on behalf of the calling process.

Tms_cutime is the sum of the *tms_utime*s and *tms_cutime*s of the child processes.

Tms_cstime is the sum of the *tms_stime*s and *tms_cstime*s of the child processes.

[EFAULT] *times* will fail if *buffer* points to an illegal address.

SEE ALSO

exec(2), fork(2), time(2), wait(2).

DIAGNOSTICS

Upon successful completion, *times* returns the elapsed real time, in clock ticks per second, from an arbitrary point in the past (e.g., system start-up time). This point does not change from one invocation of *times* to another. If *times* fails, a –1 is returned and *errno* is set to indicate the error.

NAME

truncate, ftruncate – truncate a file to a specified length

SYNOPSIS

truncate (path, length)
char *path;
long length;

ftruncate (fd, length)
int fd;
long length;

DESCRIPTION

truncate causes the file named by *path* or referenced by *fd* to be truncated to at most *length* bytes in size. If the file previously was larger than this size, the extra data is lost. With *ftruncate*, the file must be open for writing. If *length* is negative, the file is truncated to zero.

truncate succeeds unless:

[ENAMETOOLONG] The length of *path* exceeds *{PATH_MAX}*, or a pathname component is longer than *{NAME_MAX}*.

[ENOTDIR] A component of the path prefix of *path* is not a directory.

[ENOENT] The named file does not exist.

[EACCES] A component of the *path* prefix denies search permission.

[EISDIR] The named file is a directory.

[EROFS] The named file resides on a read-only file system.

[ETXTBSY] The file is a pure procedure (shared text) file that is being executed.

[EFAULT] *path* points outside the process's allocated address space.

ftruncate succeeds unless:

[EBADF] The *fd* is not a valid descriptor.

DIAGNOSTICS

A value of 0 is returned if the call succeeds. If the call fails a −1 is returned, and the global variable *errno* specifies the error.

SEE ALSO

open(2)

BUGS

Partial blocks discarded as the result of truncation are not zero filled; this can result in holes in files which do not read as zero.

These calls should be generalized to allow ranges of bytes in a file to be discarded.

NAME

uadmin – administrative control

C SYNOPSIS

#include <sys/uadmin.h>

int uadmin (cmd, fcn, mdep)
int cmd, fcn, mdep;

DESCRIPTION

uadmin provides control for basic administrative functions. This system call is tightly coupled to the system administrative procedures and is not intended for general use. The argument *mdep* is provided for machine-dependent use and is not defined here.

As specified by *cmd*, the following commands are available:

A_SHUTDOWN The system is shutdown. All user processes are killed, the buffer cache is flushed, and the root file system is unmounted. The action to be taken after the system has been shut down is specified by *fcn*. The functions are generic; the hardware capabilities vary on specific machines.

- AD_HALT Halt the processor and turn off the power.
- AD_BOOT Reboot the system, using /unix.
- AD_IBOOT Interactive reboot; user is prompted for system name. Not supported; it is treated the same as AD_HALT.

A_REBOOT The system stops immediately without any further processing. The action to be taken next is specified by *fcn* as above.

A_REMOUNT The root file system is mounted again after having been fixed. This should be used only during the startup process.

A_KILLALL All processes are killed except those belonging to the process group specified by *fcn*. They are sent the signal specified by *mdep*.

uadmin fails if any of the following are true:

[EPERM] The effective user ID is not super-user.

DIAGNOSTICS

Upon successful completion, the value returned depends on *cmd* as follows:

A_SHUTDOWN	Never returns.
A_REBOOT	Never returns.
A_REMOUNT	0
A_KILLALL	0

Otherwise, a value of -1 is returned and *errno* is set to indicate the error.

NAME

ulimit – get and set user limits

C SYNOPSIS

long ulimit (cmd, newlimit)
int cmd;
long newlimit;

DESCRIPTION

This function provides for control over process limits. The *cmd* values available are:

1 Get the regular file size limit of the process. The limit is in units of 512-byte blocks and is inherited by child processes. Files of any size can be read.

2 Set the regular file size limit of the process to the value of *newlimit*. Any process may decrease this limit, but only a process with an effective user ID of super-user may increase the limit. *ulimit* fails and the limit is unchanged if a process with an effective user ID other than super-user attempts to increase its regular file size limit. [EPERM]

3 Get the maximum possible break value [see *brk*(2)].

4 Get the current value of the maximum number of open files per process configured in the system.

SEE ALSO

brk(2), setrlimit(2), write(2).

WARNING

ulimit is effective in limiting the growth of regular files. Pipes are currently limited to 10240 bytes.

DIAGNOSTICS

Upon successful completion, a non-negative value is returned. Otherwise, a value of −1 is returned and *errno* is set to indicate the error.

NAME

umask – set and get file creation mask

C SYNOPSIS

#include <sys/types.h>
#include <sys/stat.h>

mode_t umask (mode_t cmask);

DESCRIPTION

umask sets the process's file mode creation mask to *cmask* and returns the previous value of the mask. Only the low-order 9 bits of *cmask* and the file mode creation mask are used.

SEE ALSO

chmod(2), creat(2), mknod(2), open(2).
mkdir(1), sh(1) in the *User's Reference Manual.*

DIAGNOSTICS

The previous value of the file mode creation mask is returned.

NAME

umount – unmount a file system

C SYNOPSIS

int umount (file)
char *file;

DESCRIPTION

umount requests that a previously mounted file system contained on the block special device or directory identified by *file* be unmounted. *File* is a pointer to a path name. After unmounting the file system, the directory upon which the file system was mounted reverts to its ordinary interpretation.

umount may be invoked only by the super-user.

umount will fail if one or more of the following are true:

[EPERM]	The process's effective user ID is not super-user.
[EINVAL]	*File* does not exist.
[ENOTBLK]	*File* is not a block special device.
[EINVAL]	*File* is not mounted.
[EBUSY]	A file on *file* is busy.
[EFAULT]	*File* points to an illegal address.

SEE ALSO

mount(2).

DIAGNOSTICS

Upon successful completion a value of 0 is returned. Otherwise, a value of −1 is returned and *errno* is set to indicate the error.

NAME

uname – get identity of current IRIX system

SYNOPSIS

#include <sys/utsname.h>

int uname (struct utsname *name);

DESCRIPTION

uname stores information identifying the current IRIX system in the structure pointed to by *name*.

uname uses the structure defined in **<sys/utsname.h>** whose members are:

```
char    sysname[9];
char    nodename[9];
char    release[9];
char    version[9];
char    machine[9];
```

Upon a successful return from *uname*, *sysname* contains a null-terminated character string naming the current IRIX system. Similarly, *nodename* contains the name that the system is known by on a communications network.

Release and *version* identify the operating system release and version. *Release* has one of the following forms: *m.n* or *m.n.a* where *m* is the major release number, *n* is the minor release number and *a* is the (optional) maintenance level of the release; e.g. **3.2** or **3.2.1**. *Version* contains the date and time that the operating system was generated. It has the form: *mmddhhmm*.

Machine contains the type of CPU board that the IRIX system is running on, e.g. **IP6**.

uname will fail if the following is true:

[EFAULT] *name* points to an invalid address.

SEE ALSO

gethostname(2).
uname(1) in the *User's Reference Manual*.

DIAGNOSTICS

Upon successful completion, a non-negative value is returned. Otherwise, a value of –1 is returned and *errno* is set to indicate the error.

BUGS

Nodename is limited to 8 characters, when in fact the hostname of a given system (which is the same as the nodename) may be up to 64 characters. Thus *uname* will give back truncated information. Use *gethostname*(2) instead.

NAME

unlink – remove directory entry

C SYNOPSIS

#include <unistd.h>

int unlink (const char *path);

DESCRIPTION

unlink removes the directory entry named by the path name pointed to by *path*.

The named file is unlinked unless one or more of the following are true:

[ENOTDIR]	A component of the path prefix is not a directory.
[ENOENT]	The named file does not exist.
[EACCES]	Search permission is denied for a component of the path prefix.
[EACCES]	Write permission is denied on the directory containing the link to be removed.
[EACCES]	The parent directory has the sticky bit set and the file is not writable by the user and the user does not own the parent directory and the user does not own the file and the user is not superuser.
[EPERM]	The named file is a directory and the effective user ID of the process is not super-user.
[ENAMETOOLONG]	The length of *path* exceeds *{PATH_MAX}*, or a path-name component is longer than *{NAME_MAX}*.
[ELOOP]	Too many symbolic links were encountered in translating the pathname.
[EBUSY]	The entry to be unlinked is the mount point for a mounted file system.
[EROFS]	The directory entry to be unlinked is part of a read-only file system.
[EFAULT]	*Path* points outside the process's allocated address space.

When all links to a file have been removed and no process has the file open, the space occupied by the file is freed and the file ceases to exist. If one or more processes have the file open when the last link is removed, the removal is postponed until all references to the file have been closed.

SEE ALSO

close(2), link(2), open(2), rename(2), rmdir(2).
rm(1) in the *User's Reference Manual.*

DIAGNOSTICS

Upon successful completion, a value of 0 is returned. Otherwise, a value of −1 is returned and *errno* is set to indicate the error.

NAME

ustat – get file system statistics

SYNOPSIS

```
#include <sys/types.h>
#include <ustat.h>

int ustat (dev, buf)
dev_t dev;
struct ustat *buf;
```

DESCRIPTION

ustat returns information about a mounted file system. *Dev* is a device number identifying a device containing a mounted file system. *Buf* is a pointer to a *ustat* structure that includes the following elements:

```
daddr_t  f_tfree;      /* Total free blocks */
ino_t    f_tinode;     /* Number of free inodes */
char     f_fname[6];   /* Filsys name */
char     f_fpack[6];   /* Filsys pack name */
```

ustat will fail if one or more of the following are true:

[EINVAL] *Dev* is not the device number of a device containing a mounted file system.

[EFAULT] *Buf* points outside the process's allocated address space.

[EINTR] A signal was caught during a *ustat* system call.

SEE ALSO

stat(2), statfs(2), fs(4).

DIAGNOSTICS

Upon successful completion, a value of 0 is returned. Otherwise, a value of −1 is returned and *errno* is set to indicate the error.

NOTES

This call is obsolete, use *statfs*(2) instead.

NAME

utime – set file access and modification times

SYNOPSIS

#include <unistd.h>

int utime (const char *path, struct utimbuf *times);

DESCRIPTION

Path points to a path name naming a file. *utime* sets the access and modification times of the named file.

If *times* is NULL, the access and modification times of the file are set to the current time. A process must be the owner of the file or have write permission to use *utime* in this manner.

If *times* is not NULL, *times* is interpreted as a pointer to a *utimbuf* structure (defined in <utime.h>) and the access and modification times are set to the values contained in the designated structure. Only the owner of the file or the super-user may use *utime* this way.

The times in the following structure are measured in seconds since 00:00:00 GMT, Jan. 1, 1970.

```
struct   utimbuf {
         time_t  actime;     /* access time */
         time_t  modtime;    /* modification time */
};
```

utime will fail if one or more of the following are true:

[ENOENT] The named file does not exist.

[ENOTDIR] A component of the path prefix is not a directory.

[EACCES] Search permission is denied by a component of the path prefix.

[EPERM] The effective user ID is not super-user and not the owner of the file and *times* is not NULL.

[EACCES] The effective user ID is not super-user and not the owner of the file and *times* is NULL and write access is denied.

[ENAMETOOLONG] The length of *path* exceeds *{PATH_MAX}*, or a pathname component is longer than *{NAME_MAX}*.

[EROFS] The file system containing the file is mounted read-only.

[EFAULT] *Times* is not NULL and points outside the process's allocated address space.

[EFAULT] *Path* points outside the process's allocated address space.

SEE ALSO

stat(2).

DIAGNOSTICS

Upon successful completion, a value of 0 is returned. Otherwise, a value of −1 is returned and *errno* is set to indicate the error.

NAME

vfork – spawn new process in a virtual memory efficient way

SYNOPSIS

pid = vfork()
int pid;

DESCRIPTION

vfork can be used to create new processes without fully copying the address space of the old process, which can be inefficient in a paged environment. It is useful when the purpose of *fork*(2) would have been to create a new system context for an *exec*. *vfork* differs from *fork* in that the child borrows the parent's memory and thread of control until a call to *exec* (2) or an exit (either by a call to *exit*(2) or abnormally.) The parent process is suspended while the child is using its resources.

vfork returns 0 in the child's context and (later) the pid of the child in the parent's context.

vfork can normally be used just like *fork*. It does not work, however, for the child to return from the procedure that called *vfork* since the eventual return of the parent from *vfork* would then return to a no longer existent stack frame. Be careful, also, to call *_exit* rather than *exit*, should the *exec* fail, since *exit* will flush and close standard I/O channels, and thereby interfere with the parent processes standard I/O data structures. (Even with *fork* it is wrong to call *exit* since buffered data would then be flushed twice.)

SEE ALSO

fork(2), sproc(2), exec(2), wait(2),

DIAGNOSTICS

Same as for *sproc*.

BUGS

In IRIX, vfork is emulated with the *sproc*(2) system call. The Berkeley semantics state that to avoid a possible deadlock situation, processes that are children in the middle of a *vfork* are never sent SIGTTOU or SIGTTIN signals; rather, output or *ioctl*s are allowed and input attempts result in an end-of-file indication. These semantics are not upheld in the current IRIX implementation.

NAME

vhangup – virtually ''hangup'' the current control terminal

SYNOPSIS

vhangup()

DESCRIPTION

Vhangup is used by the initialization process *init*(1M) (among others) to arrange that users are given ''clean'' terminals at login, by revoking access of the previous users' processes to the terminal. To effect this, *vhangup* searches the system tables for references to the control terminal of the invoking process, revoking access permissions on each instance of the terminal that it finds. Further attempts to access the terminal by the affected processes will yield i/o errors (EBADF). Finally, a hangup signal (SIGHUP) is sent to the process group of the control terminal.

SEE ALSO

init (1M)

BUGS

Access to the control terminal via **/dev/tty** is still possible.

NAME

wait, waitpid, wait3 – wait for child processes to stop or terminate

C SYNOPSIS

SysV:

#include <sys/wait.h>

int wait (int *statptr);

POSIX:

#include <sys/types.h>
#include <sys/wait.h>

pid_t wait (int *statptr);

pid_t waitpid (pid_t pid, int *statptr, int options);

BSD:

#include <sys/wait.h>

int wait (union wait *statptr);

#include <sys/wait.h>
#include <sys/resource.h>

int wait3 (union wait *statptr, int options, struct rusage *rusage);

DESCRIPTION

Wait functions: The *wait* functions suspend the calling process until one of the immediate children terminate, or until a child that is being traced stops because it has hit a break point. These system calls will return prematurely if a signal is received, and if a child process stopped or terminated prior to the call then return is immediate. If the call is successful, the process ID of a child is returned. The two versions differ in the *type* of their input parameter *(statptr)*, but the information conveyed is identical if the macros in **<sys/wait.h>** are used (see below description in the PARAMETERS section).

Wait3: *Wait3* is BSD's extension of *wait*. It provides an alternate interface for programs that must not block when collecting the status of child processes.

Waitpid: The *waitpid* function is POSIX's extension of *wait*. The *pid* argument specifies a set of child processes for which status is requested. The *waitpid* function only returns the status of a child process from this set.

PARAMETERS

Statptr (all functions): If *Statptr* is non-zero, 16 bits of information called *status* are stored in the low-order 16 bits of the location pointed to by *statptr*. *Status* can be used to differentiate between stopped and terminated child processes. If the child process terminated, *status* identifies the cause of termination and passes useful information to the parent. A more precise

definition of the *status* structure is given in <*sys/wait.h*>. *Status* is interpreted as follows:

If the child process stopped, the predicate **WIFSTOPPED**(**statptr*) will evaluate to non-zero and **WSTOPSIG**(**statptr*) will return the signal number that caused the process to stop. (The high-order 8 bits of status will contain the signal number and the low-order 8 bits are set equal to 0177.)

If the child process terminated due to an *exit* call, the predicate **WIFEXITED**(**statptr*) will evaluate to non-zero, and **WEXITSTATUS**(**statptr*) will return the argument that the child process passed to *_exit* or *exit*, or the value the child process returned from *main* [see *exit*(2)]. (The low-order 8 bits of status will be zero and the high-order 8 bits will contain the low-order 8 bits of the exiting argument.)

If the child process terminated due to a signal, the predicate **WIFSIGNALED**(**statptr*) will evaluate to non-zero, and **WTERMSIG**(**statptr*) will return the signal number that caused the termination. (The high-order 8 bits of status will be zero and the low-order 8 bits will contain the number of the signal.) In addition, if the low-order seventh bit (i.e., bit 0200) is set, a "core image" will have been produced [see *signal*(2)].

Rusage (wait3): If *wait3's rusage* parameter is non-zero, a summary of the resources used by the terminated process and all its children is returned (this information is currently not available for stopped processes).

Pid (waitpid):

1) If *pid* is equal to −1, status is requested for any child process. In this respect, *waitpid* is then equivalent to *wait*.

2) If *pid* is greater than zero, it specifies the process ID of a single child process for which status is requested.

3) If *pid* is equal to zero, status is requested for any child process whose process group ID is equal to that of the calling process.

4) If *pid* is less than −1, status is requested for any child process whose process group ID is equal to the absolute value of *pid*.

Options (waitpid and wait3): The *options argument* is constructed from the bitwise inclusive OR of zero or more of the following flags, defined in the header <*sys/wait.h*>:

WNOHANG The function will not suspend execution of the calling process if status is not immediately available for one of the child processes.

WUNTRACED The status of child processes that are stopped due to a **SIGTTIN**, **SIGTTOU**, **SIGTSTP**, or **SIGSTOP** signal, and whose status has not yet been reported since they stopped, are reported to the requesting process.

If a parent process terminates without waiting for its child processes to terminate, the parent process ID of each child process is set to 1. This means the initialization process inherits the child processes [see *intro*(2)].

SIGCLD HANDLING

IRIX has three distinct version of signal routines: System V (*signal*(2) and *sigset*(2)), 4.3BSD (*signal*(3B) and *sigvec*(3B)), and POSIX (*sigaction*(2)). Each version has a method by which a parent can be certain that it waits on all of its children even if they are executing concurrently. In each version, the parent installs a signal handler for **SIGCLD** to wait for its children, but the specific code differs in subtle, albeit vital, ways. Sample programs below are used to illustrate each of the three methods.

Note that System V refers to this signal as **SIGCLD**, whereas BSD calls it **SIGCHLD**. For compatibility with both systems they are defined to be the same signal number, and may therefore be used interchangeably.

System V: System V's **SIGCLD** mechanism guarantees that no **SIGCLD** signals will be lost. It accomplishes this by forcing the process to reinstall the handler (via *signal* or *sigset* calls) when leaving the handler. Note that whereas *signal*(2) sets the signal disposition back to **SIG_DFL** each time the handler is called, *sigset*(2) keeps it installed, so **SIGCLD** is the only signal that demands this reinstallation, and that only because the installation call allows the kernel to check for additional instances of the signal that occurred while the process was executing in the handler. The code below is the System V example. Note that the *sigpause*(2) creates a window during which **SIGCLD** is not blocked, allowing the parent to enter its handler.

```
/*
 * System V example of wait-in-SIGCLD-handler usage
 */
#include <signal.h>
#include <stdio.h>
#include <sys/wait.h>

static void handler(int);

#define NUMKIDS   4
```

```
volatile int kids = NUMKIDS;

main()
{
    int i, pid;

    sigset(SIGCLD, handler);
    sighold(SIGCLD);
    for (i = 0; i < NUMKIDS; i++) {
        if (fork() == 0) {
            printf("Child %d\n", getpid());
            exit(0);
        }
    }
    while (kids > 0) {
        sigpause(SIGCLD);
        sighold(SIGCLD);
    }
}

static void
handler(int sig)
{
    int pid, status;

    printf("Parent (%d) in handler, ", getpid());
    pid = wait(&status);
    kids--;
    printf("child %d, now %d left\n", pid, kids);
    /*
     * Now reinstall handler & cause SIGCLD to be re-raised
     * if any more children exited while we were in here.
     */
    sigset(SIGCLD, handler);
}
```

BSD: 4.3BSD solved this problem differently: instead of guaranteeing that no **SIGCHLD** signals are lost, it provides a **WNOHANG** option to *wait3* that allows parent processes to do non-blocking waits in loops, until no more stopped or zombied children exist. Note that the handler must be able to deal with the case in which no applicable children exist; if one or more children exit while the parent is in the handler, all may get reaped, yet if one or more **SIGCHLD** signals arrived while the parent was in its handler, the signal will remain pending, the parent will reenter the handler, and the *wait3*

call will return 0. Note that it is not necessary to call *sigvec* upon exit from the handler.

```
/*
 * BSD example of wait3-in-SIGCHLD handler usage
 */

#define _BSD_SIGNALS
#include <signal.h>
#include <stdio.h>
#include <sys/wait.h>

static int handler(int);

#define NUMKIDS   4
volatile int kids = NUMKIDS;

main()
{
    int i, pid;
    struct sigvec vec;

    vec.sv_handler = handler;
    vec.sv_mask = sigmask(SIGCHLD);
    vec.sv_flags = 0;

    sigvec(SIGCHLD, &vec, NULL);
    sigsetmask(sigmask(SIGCHLD));
    for (i = 0; i < NUMKIDS; i++) {
        if (fork() == 0) {
            printf("Child %d\n", getpid());
            exit(0);
        }
    }
    while (kids > 0) {
        sigpause(0);
    }
}

static int
handler(int sig)
{
    int pid;
    union wait status;
```

```
        printf("Parent (%d) in handler, ", getpid());
        while ((pid = wait3(&status, WNOHANG, NULL)) > 0) {
            kids--;
            printf("child %d, now %d left\n", pid, kids);
        }
    }
```

POSIX: POSIX improved on the BSD method by providing *waitpid*, that allows a parent to wait on a particular child process if desired. In addition, the IRIX implementation of *sigaction*(2) checks for zombied children upon exit from the system call if the specified signal was **SIGCLD** and the disposition of the signal handling was changed. If zombied children exist, another **SIGCLD** is raised. This solves the problem that occurs when a parent creates children, but a module that it links with (typically a libc routine such as *system*(3)) creates and waits on its own children.

Two problems have classically arisen in such a scheme: 1) until the advent of *waitpid*, the called routine could not specify which children to wait on; it therefore looped, waiting and discarding children until the one (or ones) it had created terminated, and 2) if the called routine changed the disposition of **SIGCLD** and then restored the previous handler upon exit, children of the parent (calling) process that had terminated while the called routine executed would be missed in the parent, because the called routine's **SIGCLD** handler would reap and discard those children. The addition of *waitpid* and the IRIX implementation of *sigaction* solves both of these problems. Note that neither the BSD nor the System V signal routines on IRIX have these properties, in the interests of compatibility.

WARNING: programs that install **SIGCLD** handlers that set flags instead of executing *waitpid*s and then attempt to restore the previous signal handler (via *sigaction*) upon return from the handler will create infinite loops.

```
/*
* POSIX example of waitpid-in-SIGCHLD handler usage
*/

#include <signal.h>
#include <stdio.h>
#include <sys/wait.h>

static void handler(int);

#define NUMKIDS   4
volatile int kids = NUMKIDS;

/*
```

```
* If waitpid's 1st argument is -1, it waits for any child.
*/
#define ANYKID -1

main()
{
    int i;
    pid_t pid;
    struct sigaction act;
    sigset_t set, emptyset;

    act.sa_handler = handler;
    act.sa_mask = sigmask(SIGCHLD);
    act.sa_flags = 0;

    sigaction(SIGCHLD, &act, NULL);
    sigemptyset(&set);
    sigemptyset(&emptyset);
    sigaddset(&set, SIGCHLD);
    sigprocmask(SIG_BLOCK, &set, NULL);
    setbuf(stdout, NULL);

    for (i = 0; i < NUMKIDS; i++) {
        if (fork() == 0) {
            printf("Child %d\n", getpid());
            exit(0);
        }
    }
    while (kids > 0) {
        sigsuspend(&emptyset);
    }
}

static void
handler(int sig)
{
    pid_t pid;
    int status;

    printf("Parent (%d) in handler, ", getpid());
    pid = waitpid(ANYKID, &status, WNOHANG);
    while (pid > 0) {
        kids--;
```

```
            printf("child %d, now %d left\n", pid, kids);
            pid = waitpid(ANYKID, &status, WNOHANG);
        }
    }
```

DIAGNOSTICS

Wait fails and its actions are undefined if *statptr* points to an invalid address. If *wait*, *wait3*, or *waitpid* return due to a stopped or terminated child process, the process ID of the child is returned to the calling process. *Wait3* and *waitpid* return 0 if **WNOHANG** is specified and there are currently no stopped or exited children (although children DO exist). Otherwise, a value of −1 is returned and *errno* is set to indicate the error:

[EINTR] **wait, wait3, waitpid:** The calling process received a signal.

[ECHILD] **wait, wait3, waitpid:** The calling process has no existing unwaited-for child processes. **waitpid:** The process or process group specified by *pid* does not exist or is not a child of the calling process.

[EFAULT] **wait3, waitpid:** The *rusage* or *statptr* arguments (where applicable) point to illegal addresses.

[EINVAL] **waitpid:** The value of the *options* argument is not valid.

SEE ALSO

exec(2), exit(2), fork(2), intro(2), pause(2), ptrace(2), signal(2), sigset(2), sigpause(2), sigaction(2), sigsuspend(2), sigprocmask(2), signal(3B), sigvec(3B), sigpause(3B).

NOTE

Currently, *wait3* returns only the user and system time in *rusage*.

NAME

write – write on a file

C SYNOPSIS

#include <unistd.h>

int write (int fildes, const void *buf, unsigned nbyte);

DESCRIPTION

fildes is a file descriptor obtained from a *creat*(2), *open*(2), *dup*(2), *fcntl*(2), *pipe*(2), *socket*(2), or *socketpair*(2) system call.

write attempts to write *nbyte* bytes from the buffer pointed to by *buf* to the file associated with the *fildes*.

On devices capable of seeking, the actual writing of data proceeds from the position in the file indicated by the file pointer. Upon return from *write*, the file pointer is incremented by the number of bytes actually written.

On devices incapable of seeking, writing always takes place starting at the current position. The value of a file pointer associated with such a device is undefined.

If the O_APPEND flag of the file status flags is set, the file pointer will be set to the end of the file prior to each write.

For regular files, if the O_SYNC flag of the file status flags is set, *write* will not return until both the file data and file status have been physically updated. This function is for special applications that require extra reliability at the cost of performance. For block special files, if O_SYNC is set, the write will not return until the data has been physically updated.

A write to a regular file will be blocked if mandatory file/record locking is set [see *chmod*(2)], and there is a record lock owned by another process on the segment of the file to be written. If neither O_NDELAY or O_NONBLOCK are set, the write will sleep until the blocking record lock is removed, otherwise (either flag set) *write* returns –1 and *errno* is set to EAGAIN.

For STREAMS [see *intro*(2)] files, the operation of *write* is determined by the values of the minimum and maximum *nbyte* range ("packet size") accepted by the *stream.* These values are contained in the topmost *stream* module. Unless the user pushes [see I_PUSH in *streamio*(7)] the topmost module, these values can not be set or tested from user level. If *nbyte* falls within the packet size range, *nbyte* bytes will be written. If *nbyte* does not fall within the range and the minimum packet size value is zero, *write* will break the buffer into maximum packet size segments prior to sending the data downstream (the last segment may contain less than the maximum packet size). If *nbyte* does not fall within the range and the minimum value

is non-zero, *write* will fail with *errno* set to ERANGE. Writing a zero-length buffer (*nbyte* is zero) sends zero bytes with zero returned.

For STREAMS files, if O_NDELAY and O_NONBLOCK are not set and the *stream* can not accept data (the *stream* write queue is full due to internal flow control conditions), *write* will block until data can be accepted. O_NDELAY or O_NONBLOCK will prevent a process from blocking due to flow control conditions. If O_NDELAY or O_NONBLOCK is set and the *stream* can not accept data, *write* will fail, returning −1 and setting *errno* to EAGAIN. If O_NDELAY or O_NONBLOCK is set and part of the buffer has been written when a condition in which the *stream* can not accept additional data occurs, *write* will terminate and return the number of bytes written.

write will fail and the file pointer will remain unchanged if one or more of the following are true:

[EAGAIN] Mandatory file/record locking was set, O_NDELAY or O_NONBLOCK was set, and there was a blocking record lock.

[EAGAIN] Total amount of system memory available when reading via raw IO is temporarily insufficient.

[EAGAIN] Attempt to write to a *stream* that can not accept data with the O_NDELAY or O_NONBLOCK flag set.

[EBADF] *fildes* is not a valid file descriptor open for writing.

[EDEADLK] The write was going to go to sleep and cause a deadlock situation to occur.

[EFAULT] *buf* points outside the process's allocated address space.

[EFBIG] An attempt was made to write a file that exceeds the process's file size limit or the maximum file size [see *ulimit*(2)].

[EINTR] A signal was caught during the *write* system call.

[EINVAL] Attempt to write to a *stream* linked below a multiplexor.

[ENOLCK] The system record lock table was full, so the write could not go to sleep until the blocking record lock was removed.

[ENOSPC] During a *write* to an ordinary file, there is no free space left on the device.

[ENXIO] A hangup occurred on the *stream* being written to.

[EPIPE and SIGPIPE signal]
An attempt is made to write to a pipe that is not open for reading by any process.

[ERANGE] Attempt to write to a *stream* with *nbyte* outside specified minimum and maximum write range, and the minimum value is non-zero.

If a *write* requests that more bytes be written than there is room for (e.g., the *ulimit* [see *ulimit*(2) and *setrlimit*(2)] or the physical end of a medium), only as many bytes as there is room for will be written. For example, suppose there is space for 20 bytes more in a file before reaching a limit. A write of 512-bytes will return 20. The next write of a non-zero number of bytes will give a failure return (except as noted below).

If the file being written is a pipe (or FIFO) and the O_NDELAY flag of the file flag word is set, then write to a full pipe (or FIFO) will return a count of 0. If the file being written is a pipe (or FIFO) and the O_NONBLOCK flag of the file flag word is set, then write to a full pipe (or FIFO) will return −1 and set *errno* to EAGAIN. Otherwise (O_NDELAY and O_NONBLOCK clear), writes to a full pipe (or FIFO) will block until space becomes available.

A write to a STREAMS file can fail if an error message has been received at the stream head. In this case, *errno* is set to the value included in the error message.

CAVEATS

Due to the different semantics of O_NDELAY and O_NONBLOCK in the case of pipes or FIFOs, these flags *must* not be used simultaneously.

SEE ALSO

creat(2), dup(2), fcntl(2), intro(2), lseek(2), open(2), pipe(2), setrlimit(2), ulimit(2).

DIAGNOSTICS

Upon successful completion the number of bytes actually written is returned. Otherwise, −1 is returned and *errno* is set to indicate the error.